US011463314B2

(12) United States Patent
Pieczul et al.

(10) Patent No.: US 11,463,314 B2
(45) Date of Patent: Oct. 4, 2022

(54) AUTOMATICALLY INFERRING SOFTWARE-DEFINED NETWORK POLICIES FROM THE OBSERVED WORKLOAD IN A COMPUTING ENVIRONMENT

(71) Applicant: **Oracle International Corporation**, Redwood Shores, CA (US)

(72) Inventors: **Olgierd Stanislaw Pieczul**, Dublin (IE); **Robert Clark**, Clyde Hill, WA (US); **Nitin Srinivasa Rao Jami**, Mercer Island, WA (US)

(73) Assignee: **ORACLE INTERNATIONAL CORPORATION**, Redwood Shores, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **17/124,155**

(22) Filed: **Dec. 16, 2020**

(65) **Prior Publication Data**

US 2022/0191099 A1 Jun. 16, 2022

(51) **Int. Cl.**
G06F 15/177 (2006.01)
H04L 41/0893 (2022.01)
(Continued)

(52) **U.S. Cl.**
CPC ***H04L 41/0893*** (2013.01); ***H04L 41/0266*** (2013.01); ***H04L 41/0806*** (2013.01);
(Continued)

(58) **Field of Classification Search**
CPC H04L 41/0893; H04L 43/062; H04L 43/0811; H04L 41/0266
(Continued)

(56) **References Cited**

U.S. PATENT DOCUMENTS

8,429,140 B1 4/2013 Lolayekar et al.
11,102,076 B1 8/2021 Pieczul et al.
(Continued)

OTHER PUBLICATIONS

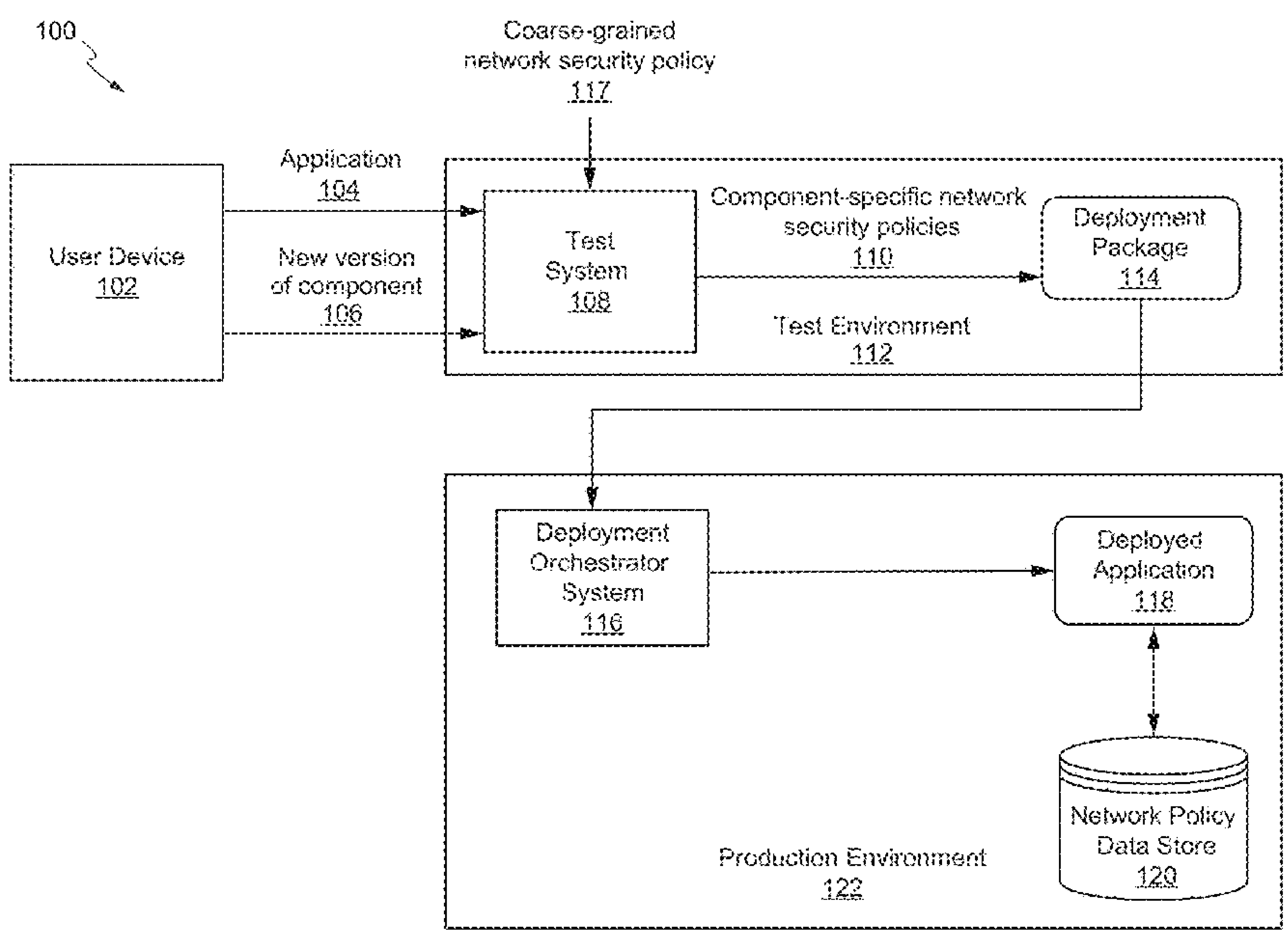
"Tufin SecureCloud Solution", tufin, the Security Policy company, May 20, 2020, 2 pages.
(Continued)

Primary Examiner — Lan Dai T Truong
(74) *Attorney, Agent, or Firm* — Kilpatrick Townsend & Stockton LLP

(57) **ABSTRACT**

Techniques are disclosed for automatically inferring software-defined network policies from the observed workload in a computing environment. The disclosed techniques include monitoring network traffic flow originating from network interfaces corresponding to containers that execute components of an application, recording details of a new network connection or a change in the existing network connection, obtaining information concerning the components of the application, identifying metadata for a component involved in the new network connection or the change in an existing network connection based on a comparison of the details of the new network connection or a change in the existing network connection and the information concerning the components of the application, generating a network policy for the component using at least the metadata for the component, and integrating the network policy for the component into a deployment package for the application.

20 Claims, 14 Drawing Sheets



(51) **Int. Cl.**

H04L 41/0266	(2022.01)
H04L 43/062	(2022.01)
H04L 41/08	(2022.01)
H04L 43/0811	(2022.01)
H04L 41/0806	(2022.01)

(52) **U.S. Cl.**
CPC ***H04L 41/0886*** (2013.01); ***H04L 43/062*** (2013.01); ***H04L 43/0811*** (2013.01)

(58) **Field of Classification Search**
USPC 709/220, 223, 224
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

2009/0199213 A1*	8/2009	Webster	G06F 9/54 719/320
2012/0093160 A1*	4/2012	Tonsing	H04L 63/0245 370/392
2012/0109958 A1	5/2012	Thakur et al.	
2012/0192246 A1	7/2012	Harrison	
2013/0139228 A1	5/2013	Odaira	
2018/0027006 A1	1/2018	Zimmermann et al.	
2019/0208445 A1	7/2019	Klatsky et al.	
2019/0289035 A1	9/2019	Ahuja et al.	
2020/0280576 A1	9/2020	Key et al.	
2022/0191099 A1	6/2022	Pieczul et al.	

OTHER PUBLICATIONS

U.S. Appl. No. 17/124,162, filed Dec. 16, 2020 in all of its entirety.

U.S. Appl. No. 17/167,591, Techniques for Network Policies Analysis in Container Frameworks filed Feb. 4, 2021, all pages.

U.S. Appl. No. 17/167,591, Notice of Allowance dated May 26, 2021, 8 pages.

U.S. Appl. No. 17/379,923, Non-Final Office Action dated Mar. 14, 2022, 9 pages.

U.S. Appl. No. 17/124,162 , "Notice of Allowance", dated Aug. 15, 2022, 8 pages.

U.S. Appl. No. 17/187,631 , "Corrected Notice of Allowability", dated Aug. 15, 2022, 5 pages.

U.S. Appl. No. 17/187,631 , "Notice of Allowance", dated Jul. 7, 2022, 12 pages.

U.S. Appl. No. 17/379,923 , "Corrected Notice of Allowability", dated Aug. 9, 2022, 2 pages.

U.S. Appl. No. 17/379,923 , "Notice of Allowance", dated Jul. 21, 2022, 5 pages.

* cited by examiner

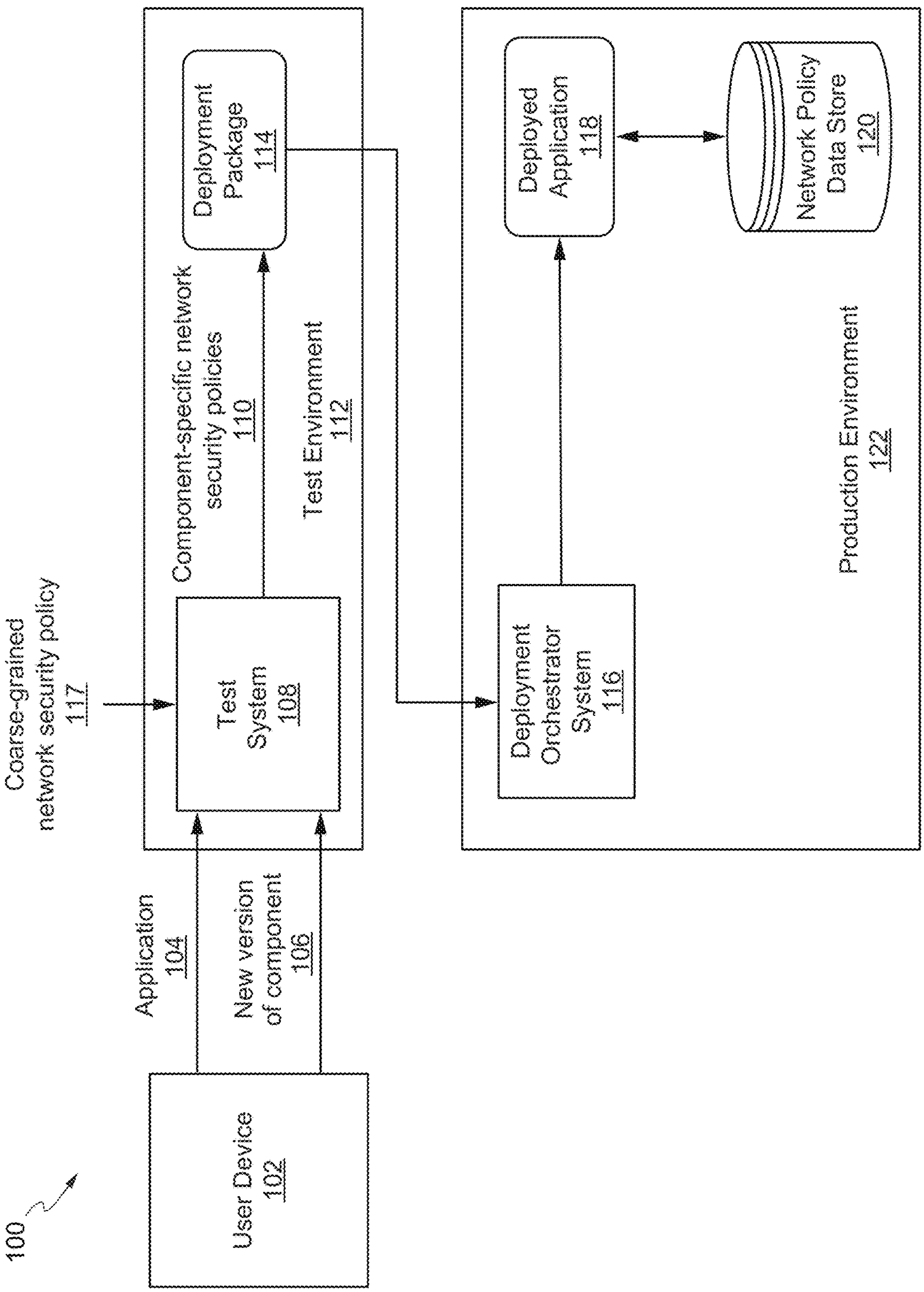

100
User Device 102
Application 104
New version of component 106
Coarse-grained network security policy 117
Test System 108
Component-specific network security policies 110
Test Environment 112
Deployment Package 114
Deployment Orchestrator System 116
Deployed Application 118
Network Policy Data Store 120
Production Environment 122

FIG. 1

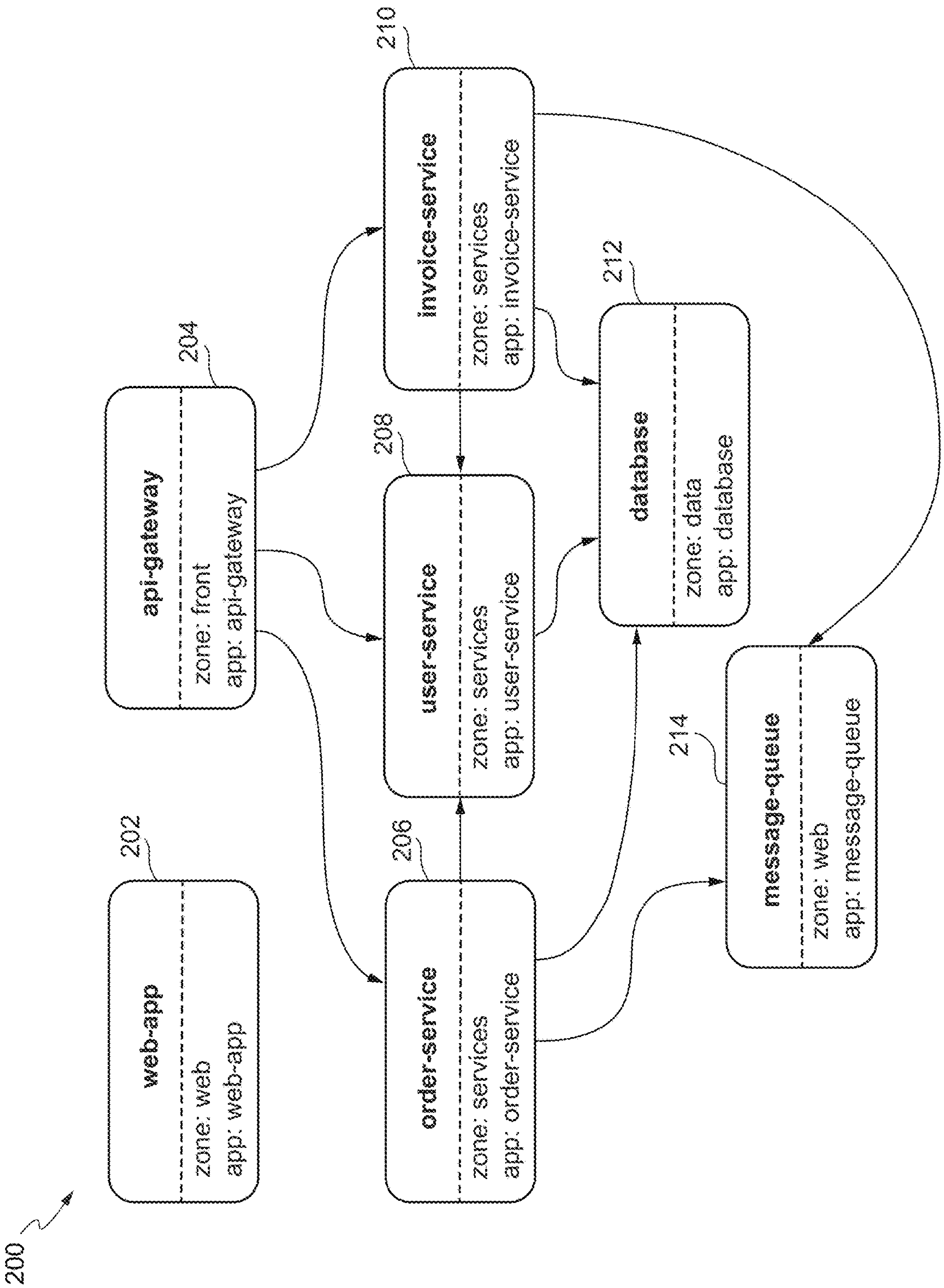
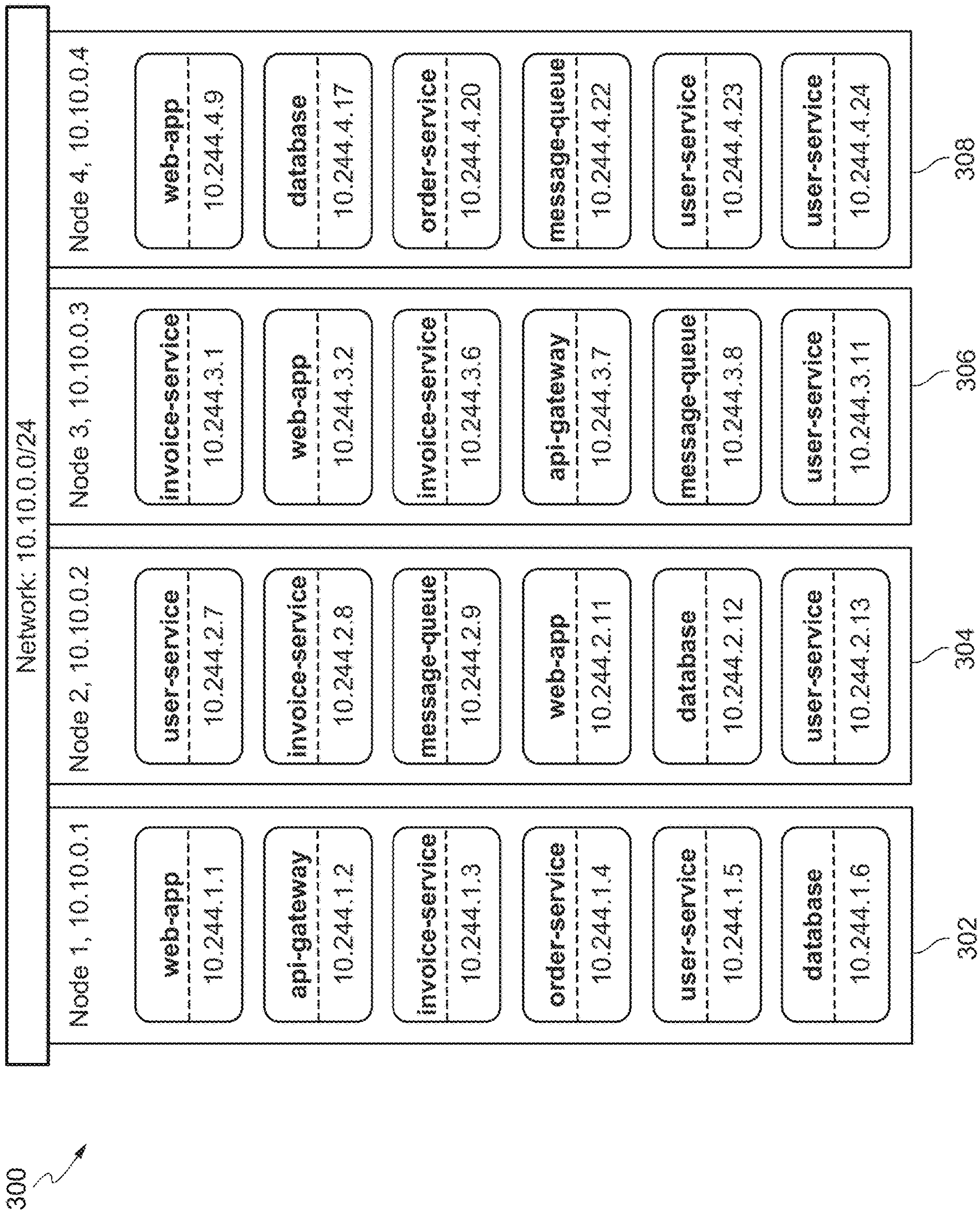

200
202
web-app
zone: web
app: web-app
204
api-gateway
zone: front
app: api-gateway
206
order-service
zone: services
app: order-service
208
user-service
zone: services
app: user-service
210
invoice-service
zone: services
app: invoice-service
212
database
zone: data
app: database
214
message-queue
zone: web
app: message-queue

FIG. 2



300
Network: 10.10.0.0/24
Node 1, 10.10.0.1
web-app
10.244.1.1
api-gateway
10.244.1.2
invoice-service
10.244.1.3
order-service
10.244.1.4
user-service
10.244.1.5
database
10.244.1.6
302
Node 2, 10.10.0.2
user-service
10.244.2.7
invoice-service
10.244.2.8
message-queue
10.244.2.9
web-app
10.244.2.11
database
10.244.2.12
user-service
10.244.2.13
304
Node 3, 10.10.0.3
invoice-service
10.244.3.1
web-app
10.244.3.2
invoice-service
10.244.3.6
api-gateway
10.244.3.7
message-queue
10.244.3.8
user-service
10.244.3.11
306
Node 4, 10.10.0.4
web-app
10.244.4.9
database
10.244.4.17
order-service
10.244.4.20
message-queue
10.244.4.22
user-service
10.244.4.23
user-service
10.244.4.24
308

FIG. 3

400

From (zone)	To (zone)
-	web
-	front
front	services
services	services
services	data

402

From (app)	To (app)
-	web-app
-	api-gateway
api-gateway	user-service
api-gateway	order-service
api-gateway	invoice-service
order-service	user-service
invoice-service	user-service
user-service	database
invoice-service	database
order-service	database
order-service	message-queue
invoice-service	message-queue

404

From	To
-	zone: web
-	zone: front
zone: front	zone: services
zone: services	zone: services
zone: services	app: database
app: order-service	app: message-queue
app: invoice-service	app: message-queue

406

FIG. 4

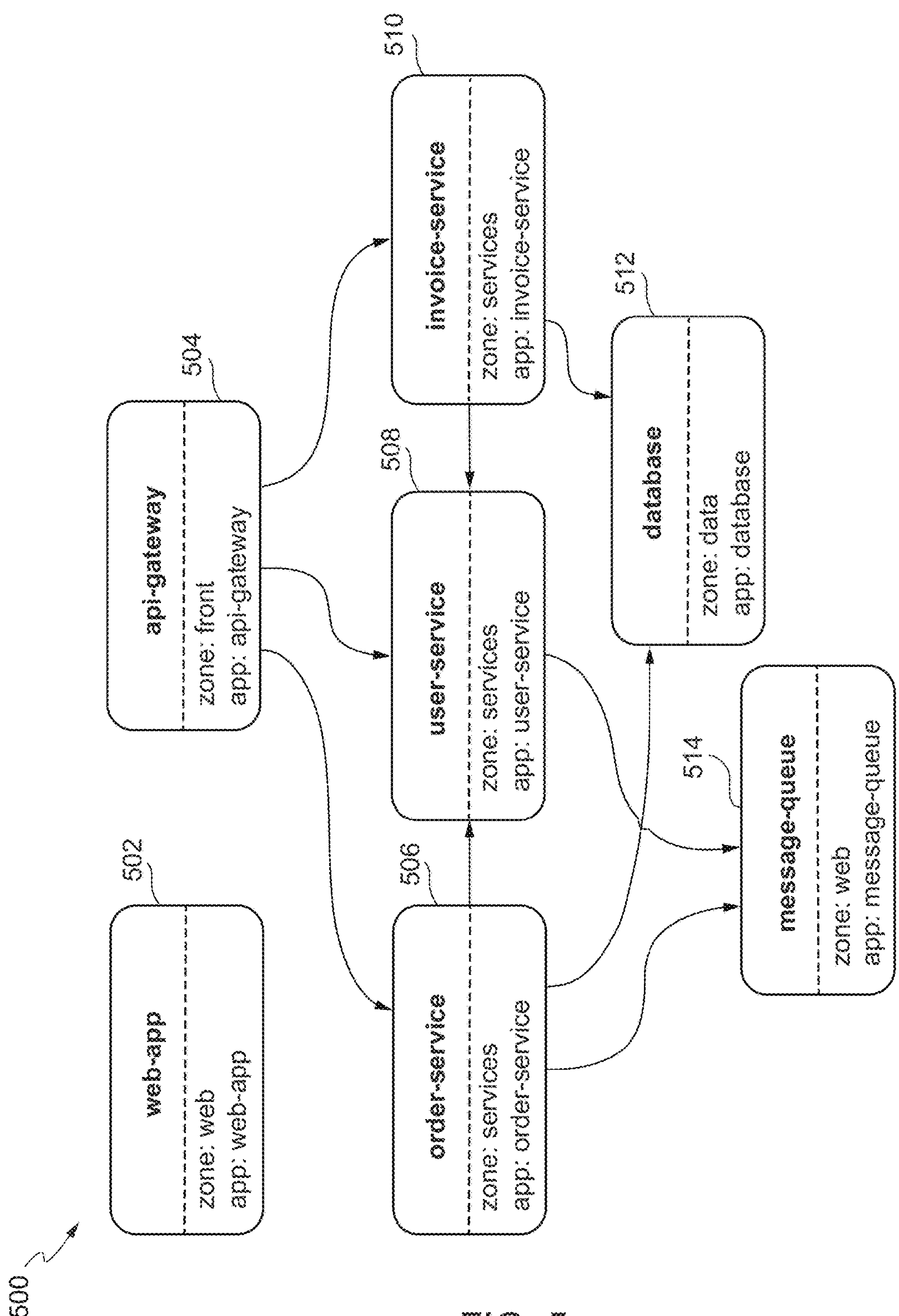

500
502
web-app
zone: web
app: web-app
504
api-gateway
zone: front
app: api-gateway
506
order-service
zone: services
app: order-service
508
user-service
zone: services
app: user-service
510
invoice-service
zone: services
app: invoice-service
512
database
zone: data
app: database
514
message-queue
zone: web
app: message-queue

FIG. 5

FIG. 5

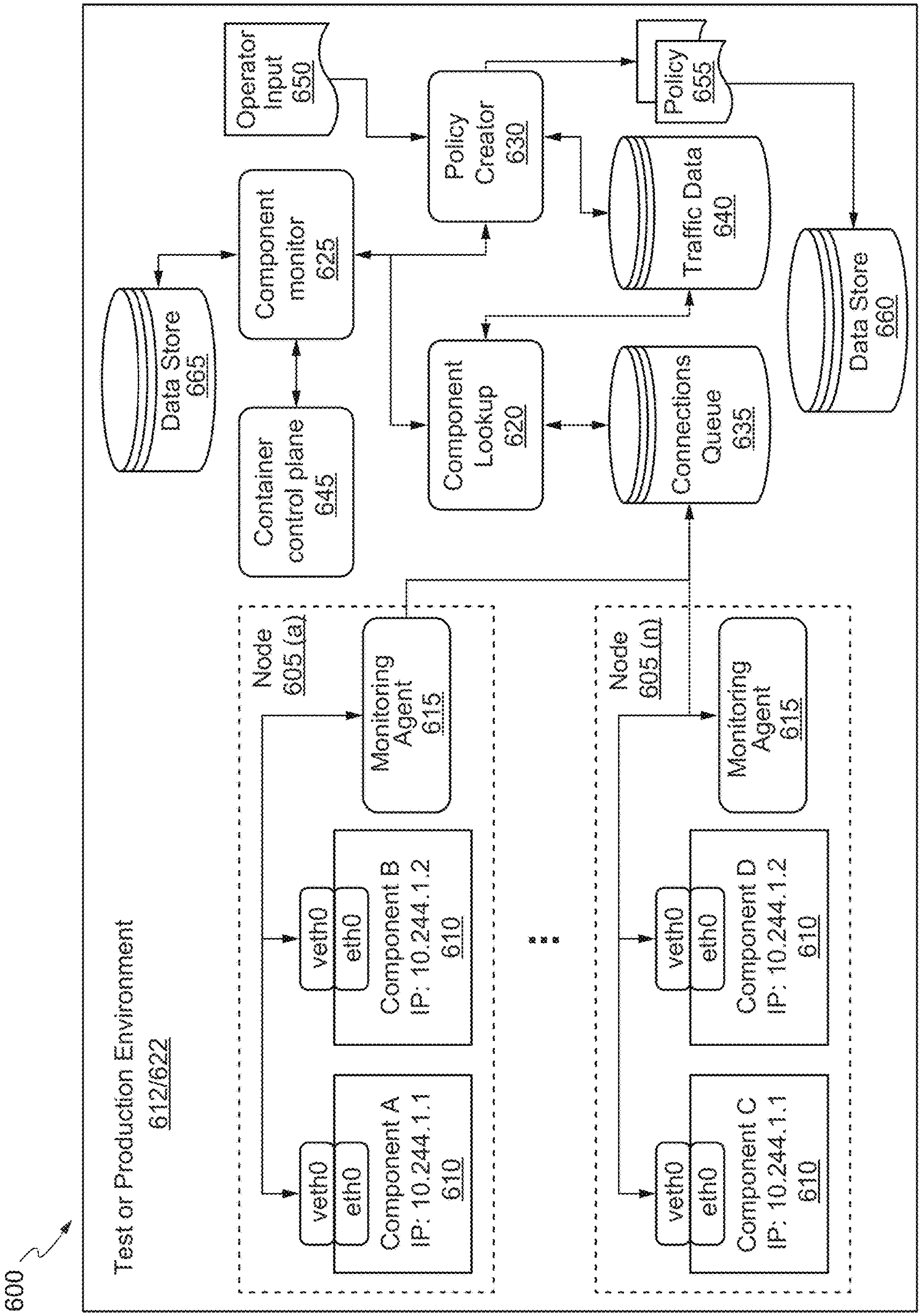

600
Test or Production Environment
612/622
Node
605 (a)
veth0
eth0
Component A
IP: 10.244.1.1
610
veth0
eth0
Component B
IP: 10.244.1.2
610
Monitoring
Agent
615
Node
605 (n)
veth0
eth0
Component C
IP: 10.244.1.1
610
veth0
eth0
Component D
IP: 10.244.1.2
610
Monitoring
Agent
615
Data Store
665
Component
monitor
625
Container
control plane
645
Operator
Input
650
Policy
Creator
630
Component
Lookup
620
Policy
655
Traffic Data
640
Connections
Queue
635
Data Store
660

FIG. 6

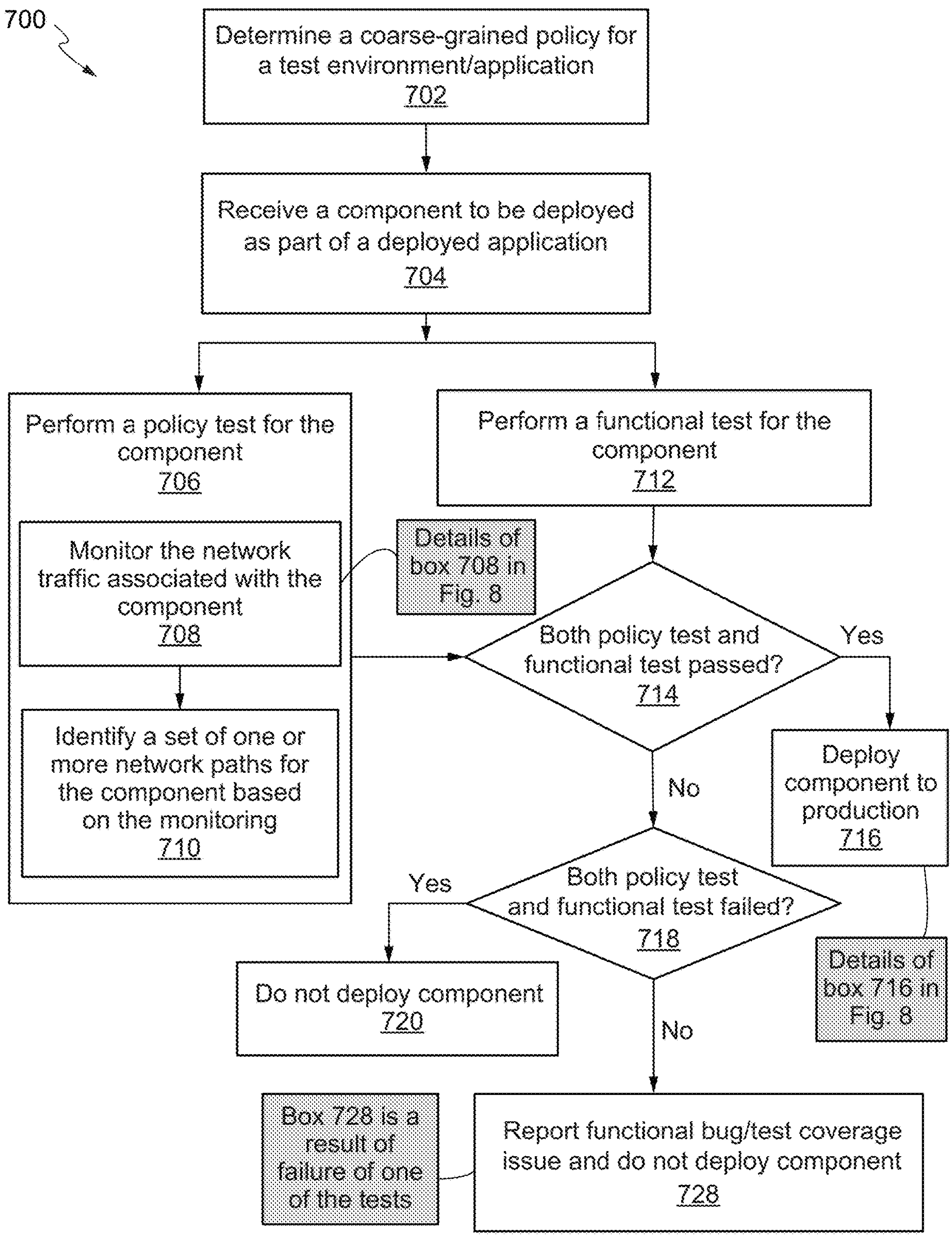

700
Determine a coarse-grained policy for a test environment/application
702
Receive a component to be deployed as part of a deployed application
704
Perform a policy test for the component
706
Monitor the network traffic associated with the component
708
Identify a set of one or more network paths for the component based on the monitoring
710
Details of box 708 in Fig. 8
Perform a functional test for the component
712
Both policy test and functional test passed?
714
Yes
No
Deploy component to production
716
Details of box 716 in Fig. 8
Both policy test and functional test failed?
718
Yes
No
Do not deploy component
720
Box 728 is a result of failure of one of the tests
Report functional bug/test coverage issue and do not deploy component
728

FIG. 7

800

Monitor network traffic
805

Detect new network or change in an existing network connection
810

Record details concerning the new network connection or the change in the existing network connection
815

Obtain information for components of the application
820

Identify metadata for a component involved in the new network connection or the change in the existing network connection based on a comparison between the record details and the information of the components
825

Generate a network policy for the component using at least the metadata for the component
830

Generate a deployment package comprising the component and the network policy associated with the component
835

Deploy the deployment package in the production environment
840

FIG. 8

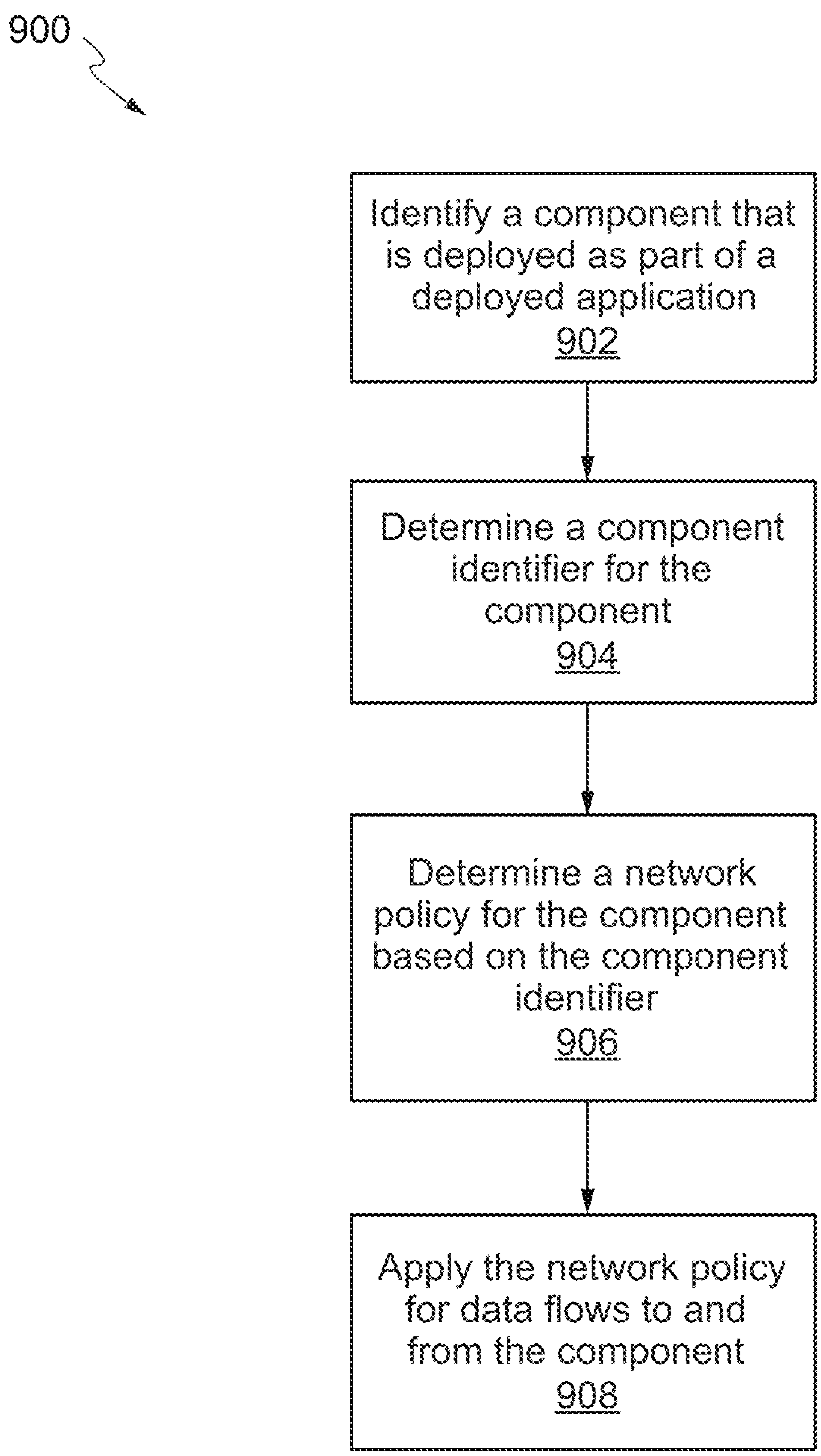
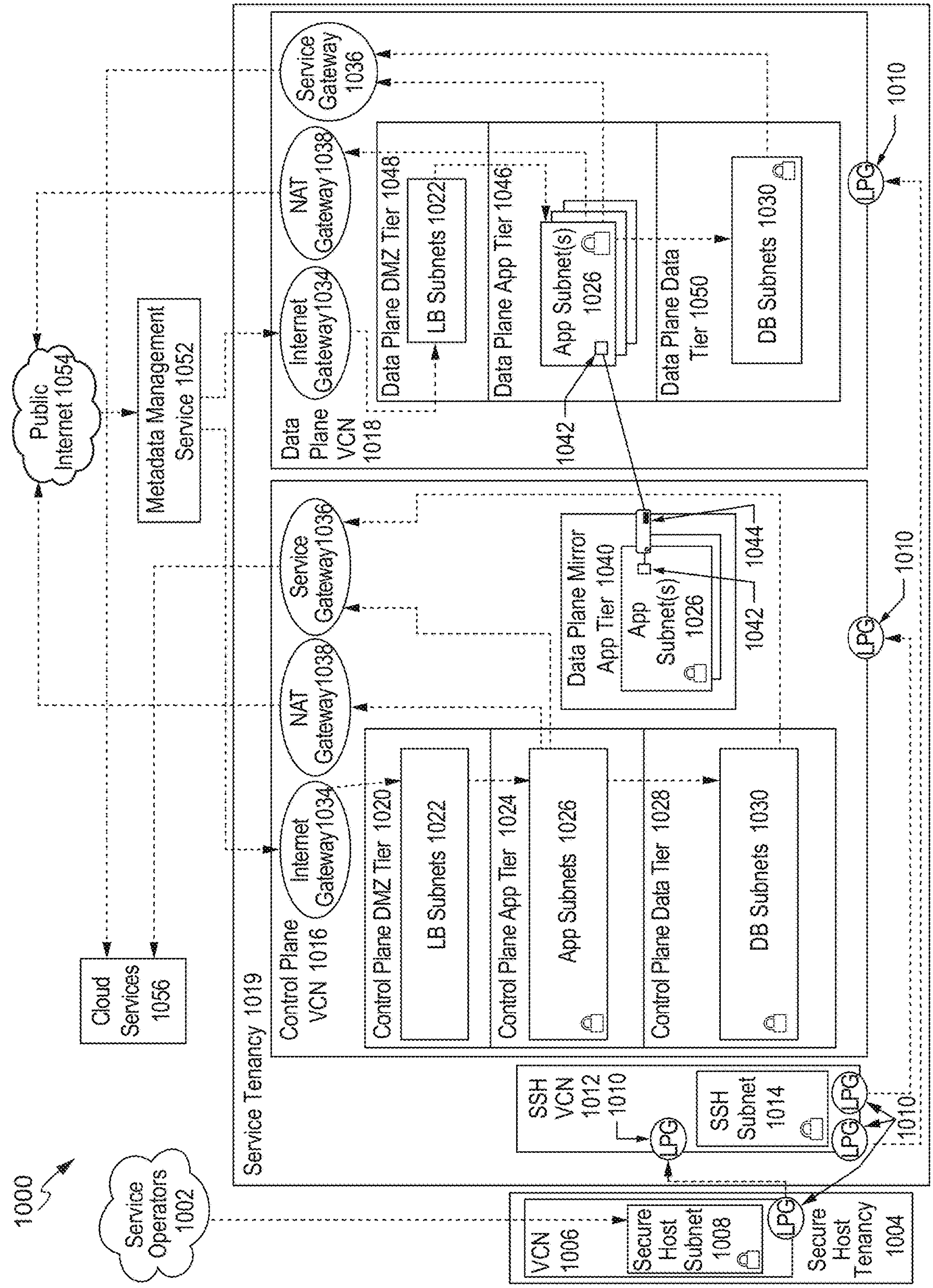
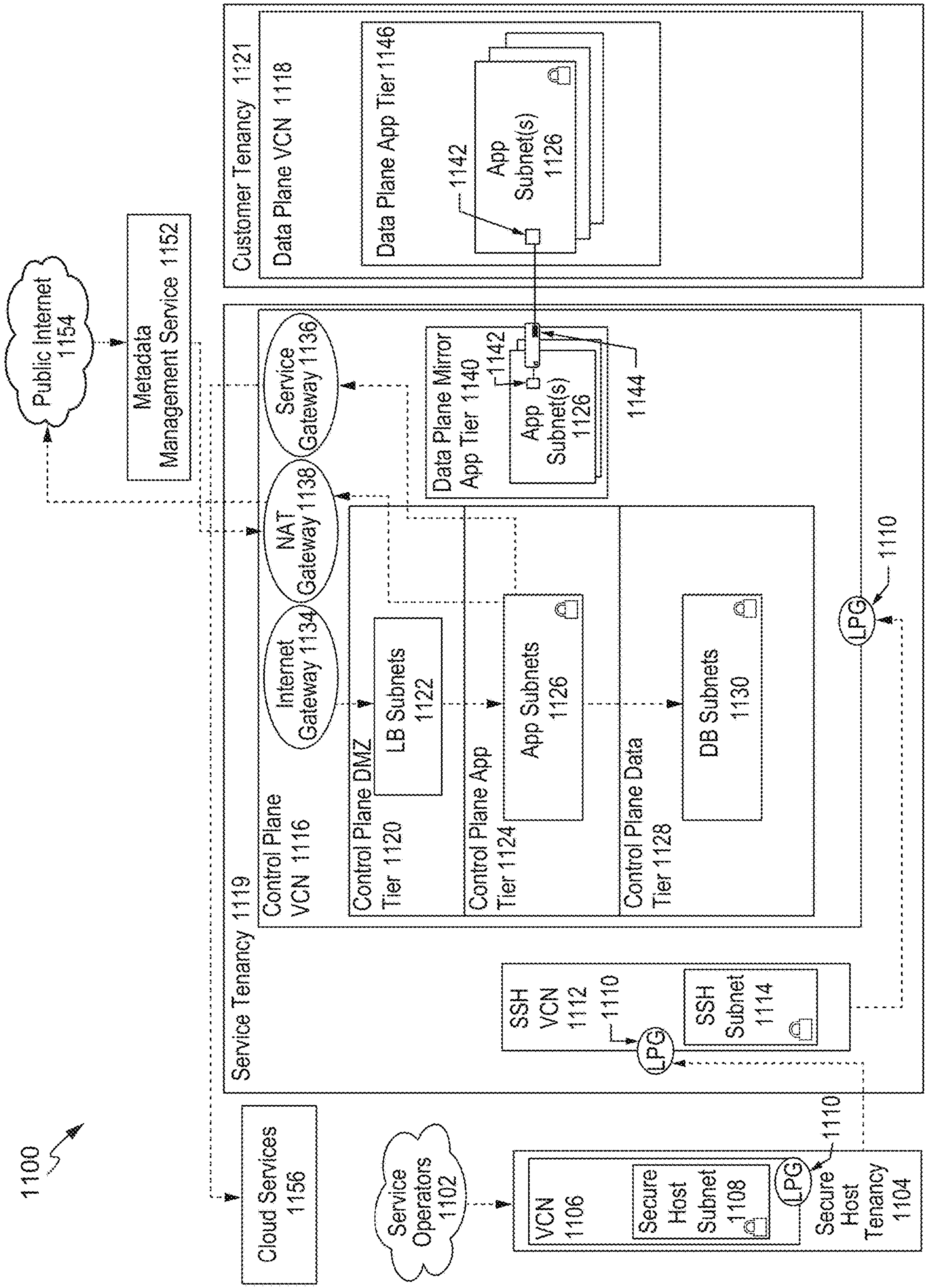

900
Identify a component that is deployed as part of a deployed application
902
Determine a component identifier for the component
904
Determine a network policy for the component based on the component identifier
906
Apply the network policy for data flows to and from the component
908

FIG. 9



1000
Service Operators 1002
Cloud Services 1056
Public Internet 1054
Metadata Management Service 1052
Service Tenancy 1019
Control Plane VCN 1016
Internet Gateway1034
NAT Gateway1038
Service Gateway1036
Control Plane DMZ Tier 1020
LB Subnets 1022
Control Plane App Tier 1024
App Subnets 1026
Control Plane Data Tier 1028
DB Subnets 1030
Data Plane Mirror App Tier 1040
App Subnet(s) 1026
1042
1044
Data Plane VCN 1018
Internet Gateway1034
NAT Gateway1038
Service Gateway 1036
Data Plane DMZ Tier 1048
LB Subnets 1022
Data Plane App Tier 1046
App Subnet(s) 1026
1042
Data Plane Data Tier 1050
DB Subnets 1030
LPG
1010
SSH VCN 1012
SSH Subnet 1014
VCN 1006
Secure Host Subnet 1008
Secure Host Tenancy 1004

FIG. 10



1100
Public Internet 1154
Metadata Management Service 1152
Cloud Services 1156
Service Operators 1102
VCN 1106
Secure Host Subnet 1108
LPG
1110
Secure Host Tenancy 1104
Service Tenancy 1119
SSH VCN 1112
SSH Subnet 1114
Control Plane VCN 1116
Control Plane DMZ Tier 1120
LB Subnets 1122
Control Plane App Tier 1124
App Subnets 1126
Control Plane Data Tier 1128
DB Subnets 1130
Internet Gateway 1134
NAT Gateway 1138
Service Gateway 1136
Data Plane Mirror App Tier 1140
App Subnet(s) 1126
1142
1144
Customer Tenancy 1121
Data Plane VCN 1118
Data Plane App Tier 1146

FIG. 11

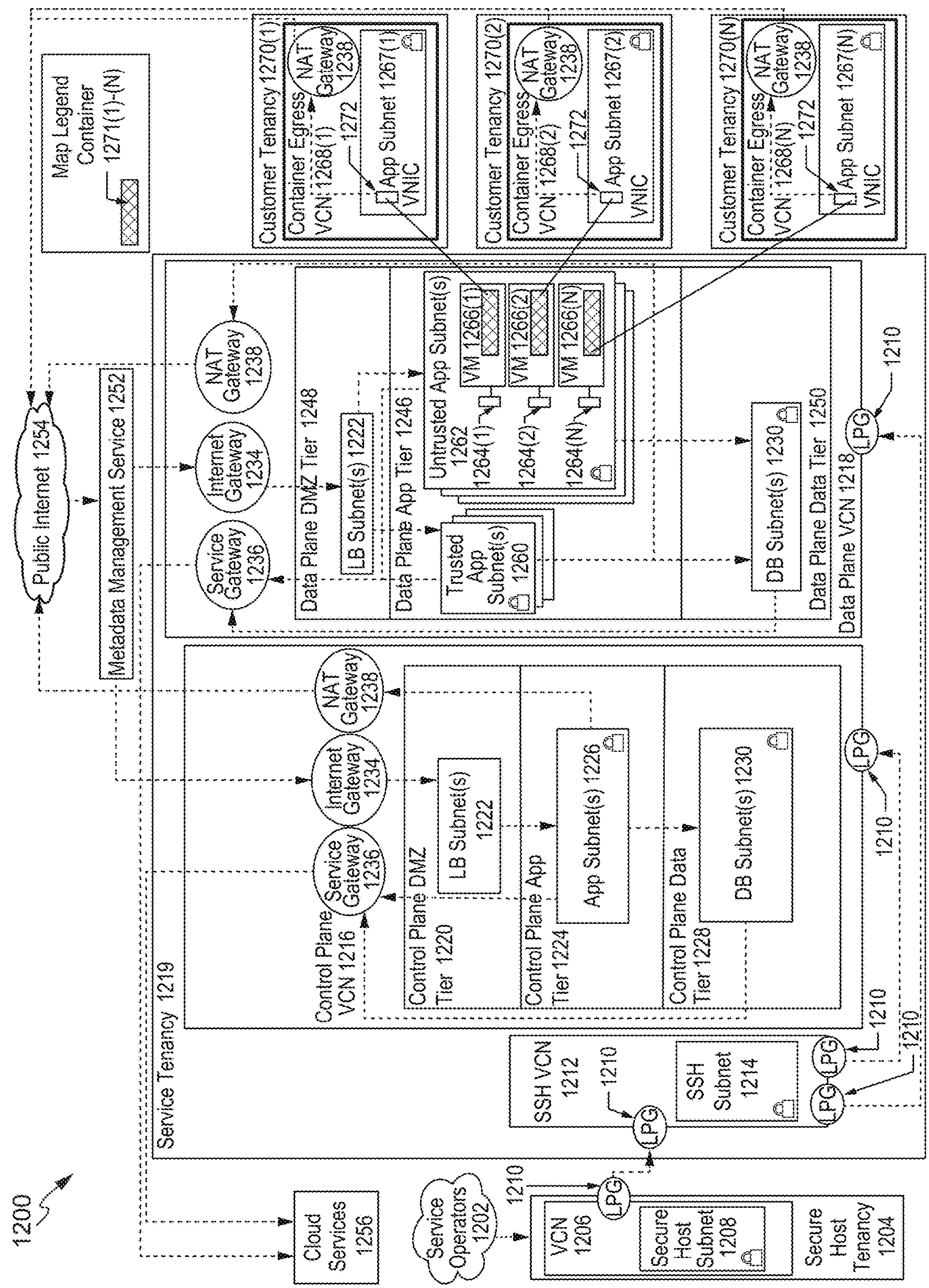

1200
Map Legend
Container
1271(1)-(N)
Customer Tenancy 1270(1)
Container Egress
VCN 1268(1)
NAT Gateway 1238
1272
App Subnet 1267(1)
VNIC
Customer Tenancy 1270(2)
Container Egress
VCN 1268(2)
App Subnet 1267(2)
Customer Tenancy 1270(N)
Container Egress
VCN 1268(N)
App Subnet 1267(N)
Public Internet 1254
Metadata Management Service 1252
Service Gateway 1236
Internet Gateway 1234
Data Plane DMZ Tier 1248
LB Subnet(s) 1222
Data Plane App Tier 1246
Trusted App Subnet(s) 1260
Untrusted App Subnet(s) 1262
VM 1266(1)
VM 1266(2)
VM 1266(N)
1264(1)
1264(2)
1264(N)
DB Subnet(s) 1230
Data Plane Data Tier 1250
Data Plane VCN 1218
LPG
1210
Control Plane VCN 1216
Control Plane DMZ Tier 1220
Control Plane App Tier 1224
App Subnet(s) 1226
Control Plane Data Tier 1228
Service Tenancy 1219
SSH VCN 1212
SSH Subnet 1214
Cloud Services 1256
Service Operators 1202
VCN 1206
Secure Host Subnet 1208
Secure Host Tenancy 1204

FIG. 12

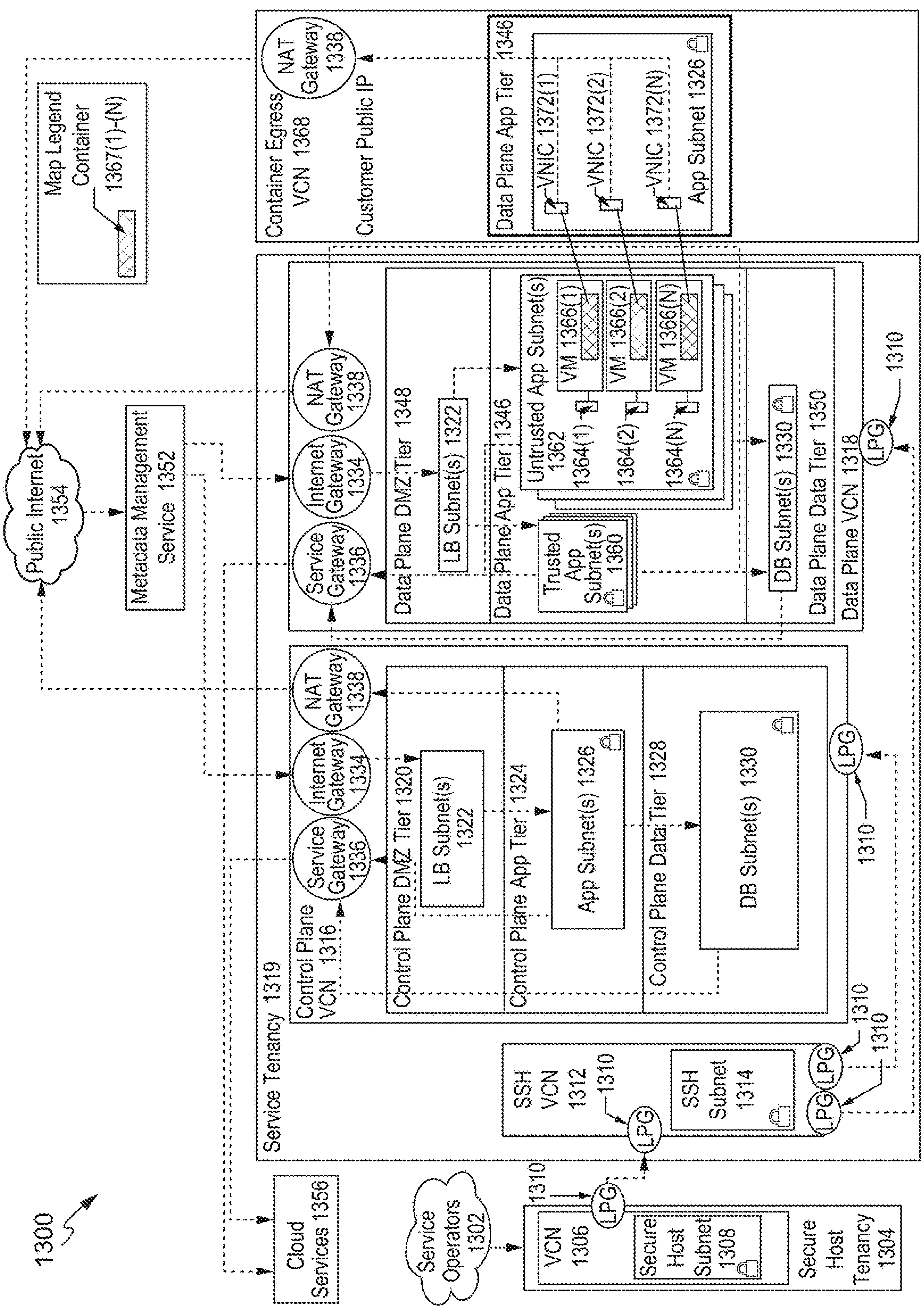

1300
Service Operators 1302
VCN 1306
Secure Host Subnet 1308
Secure Host Tenancy 1304
LPG
1310
Cloud Services 1356
SSH VCN 1312
SSH Subnet 1314
Service Tenancy 1319
Control Plane VCN 1316
Service Gateway 1336
Internet Gateway 1334
NAT Gateway 1338
Control Plane DMZ Tier 1320
LB Subnet(s) 1322
Control Plane App Tier 1324
App Subnet(s) 1326
Control Plane Data Tier 1328
DB Subnet(s) 1330
Data Plane VCN 1318
Data Plane DMZ Tier 1348
Data Plane App Tier 1346
Trusted App Subnet(s) 1360
Untrusted App Subnet(s) 1362
1364(1)
1364(2)
1364(N)
VM 1366(1)
VM 1366(2)
VM 1366(N)
Data Plane Data Tier 1350
Public Internet 1354
Metadata Management Service 1352
Map Legend Container 1367(1)-(N)
Container Egress VCN 1368
Customer Public IP
Data Plane App Tier 1346
VNIC 1372(1)
VNIC 1372(2)
VNIC 1372(N)
App Subnet 1326

FIG. 13

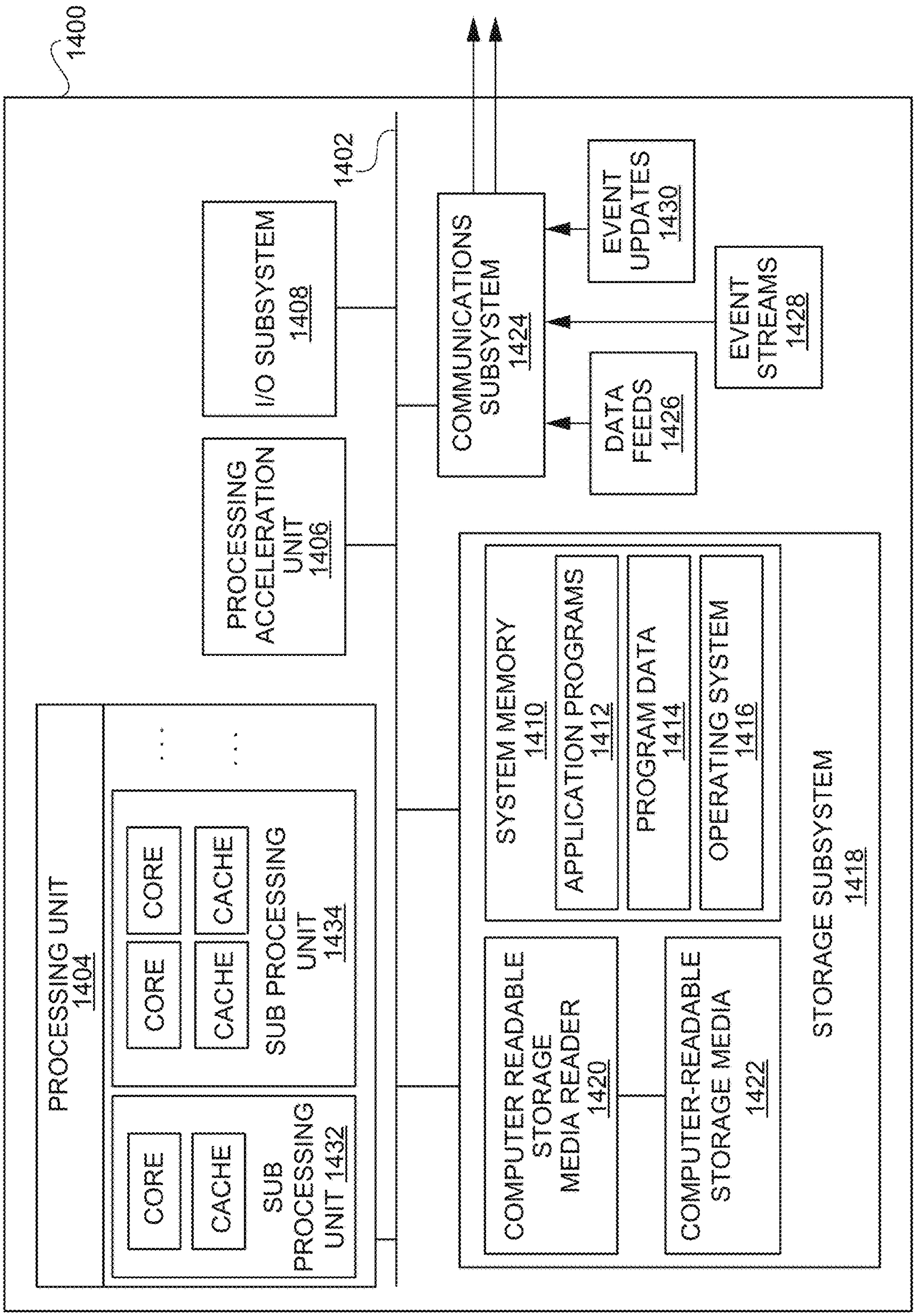

1400
1402
PROCESSING UNIT 1404
CORE
CACHE
SUB PROCESSING UNIT 1432
CORE
CORE
CACHE
CACHE
SUB PROCESSING UNIT 1434
PROCESSING ACCELERATION UNIT 1406
I/O SUBSYSTEM 1408
COMMUNICATIONS SUBSYSTEM 1424
DATA FEEDS 1426
EVENT STREAMS 1428
EVENT UPDATES 1430
SYSTEM MEMORY 1410
APPLICATION PROGRAMS 1412
PROGRAM DATA 1414
OPERATING SYSTEM 1416
COMPUTER READABLE STORAGE MEDIA READER 1420
COMPUTER-READABLE STORAGE MEDIA 1422
STORAGE SUBSYSTEM 1418

FIG.14

AUTOMATICALLY INFERRING SOFTWARE-DEFINED NETWORK POLICIES FROM THE OBSERVED WORKLOAD IN A COMPUTING ENVIRONMENT

FIELD OF THE INVENTION

The present disclosure relates generally to network traffic control, and more particularly, to techniques for automatically inferring software-defined network policies from the observed workload in a computing environment.

BACKGROUND

Container orchestration tools provide a robust framework for managing and deploying containerized applications across a cluster of computing nodes in a computing environment. Examples of these tools include, for instance, Kubernetes, Open Shift, Docker Swarm and the like. The usage of these tools has dramatically increased in the recent years with the rising popularity of cloud-based services and changes in the design of services/applications from large and monolithic systems to highly distributed and micro-service based systems. In the micro-service based model, an application is built using a large number of small components communicating over a network. Each component can be independently deployed, upgraded and scaled to a production environment. Software-defined networks are an integral part of the micro-service based model, allowing seamless changes to individual components without disruption. Each time the arrangement of a component within the system changes, the underlying network is reconfigured automatically. Components in such networks typically have dynamically assigned Internet Protocol (IP) addresses that are not stable for a particular component type.

Due to the dynamic nature of the network, existing techniques (e.g., host or network based firewalls) used by container orchestration frameworks for implementing network traffic flow controls are oftentimes inadequate. Since the network traffic leaving the host or traversing the network is often encapsulated, it is very challenging to distinguish traffic at the level of individual components that reside on the hosts (computing nodes). Even if additional logic is introduced to de-encapsulate the packets traversing the network, filtering the network traffic still poses a challenge due to the dynamic nature of the source and destination addresses of the components.

BRIEF SUMMARY

This disclosure relates generally to virtual networking environments and network traffic control within those virtual networking environments. More specifically, but not by way of limitation, this disclosure describes techniques (e.g., a method, a system, non-transitory computer-readable medium storing code or instructions executable by one or more processors) automatically inferring software-defined network policies from the observed workload in a computing environment.

In various embodiments, a method is provided that comprises: monitoring, by a data processing system, network traffic flow originating from network interfaces corresponding to containers that execute components of an application; detecting, by the data processing system, a new network connection or a change in an existing network connection within the network traffic based on the monitoring of the network traffic flow; in response to detecting the new network connection or the change in the existing network connection, recording, by the data processing system, details of the new network connection or the change in the existing network connection, where the details include a network address of a source component and a network address of a destination component for the new network connection or the change in the existing network connection; obtaining, by the data processing system, information concerning the components of the application, where the information includes the network address and metadata associated with each of the components of the application; identifying, by the data processing system, metadata for the source component and the destination component based on a comparison of at least the network address of the source component and the network address of the destination component to the network address associated with each of the components of the application; generating, by the computer system, a network policy for the source component or the destination component using at least the metadata for the source component and the destination component, where the network policy comprises information representative of the new network connection or the change in the existing network connection; and integrating, by the computer system, the network policy for the source component or the destination component into a deployment package for the application.

In some embodiments, the details further include a time stamp for the new network connection or the change in the existing network connection.

In some embodiments, the information further includes any changes to arrangement of the components and time of the changes, and the metadata comprises labels associated with each of the components of the application.

In some embodiments, the network policy for the source component or the destination component is generated using at least the metadata for the source component and the destination component, the time stamp for the new network connection or the change in the existing network connection, and the time of the changes associated with the arrangement of the source component or the destination component.

In some embodiments, the method further comprises: identifying, by the data processing system, a subset of components of the components of the application that are not involved in the new network connection or the change in the existing network connection based on the comparison of at least the network address of the source component and the network address of the destination component to the network address associated with each of the components of the application, where the network policy for the source component or the destination component is generated using at least the metadata for the source component and the destination component and the subset of components of the components of the application that are not involved in the new network connection or the change in the existing network connection.

In some embodiments, the network policy comprises information indicating that the network policy is applicable to a defined version of the source component or the destination component.

In some embodiments, the method further comprises: deploying, by the data processing system, the deployment package to a computing node in a computing environment of the data processing system.

In various embodiments, a system is provided that includes one or more data processors and a non-transitory computer readable storage medium containing instructions which, when executed on the one or more data processors, cause the one or more data processors to perform part or all of one or more methods disclosed herein.

In various embodiments, a computer-program product is provided that is tangibly embodied in a non-transitory machine-readable storage medium and that includes instructions configured to cause one or more data processors to perform part or all of one or more methods disclosed herein.

The techniques described above and below may be implemented in a number of ways and in a number of contexts. Several example implementations and contexts are provided with reference to the following figures, as described below in more detail. However, the following implementations and contexts are but a few of many.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** depicts an example of a distributed computing environment for generating network policies for components of an application deployed in the computing environment, according to various embodiments.

FIG. **2** depicts an example of an application deployed in a containerized environment of the computing environment shown in FIG. **1**, according to various embodiments.

FIG. **3** depicts an example arrangement of components of the application shown in FIG. **2** on a cluster of nodes in a container-based framework, according to various embodiments.

FIG. **4** depicts various examples of network polices defined by a container-based framework for a containerized application deployed on a cluster of nodes in the container-based framework, according to various embodiments.

FIG. **5** is an example of the containerized application shown in FIG. **2** depicting a change in the operation of one of the components within the containerized application, according to various embodiments.

FIG. **6** depicts additional details of the operations performed by the systems shown in FIG. **1** for generating network policies for components of an application deployed in a computing environment, according to various embodiments.

FIG. **7** is an example of a process for generating a network policy for a component of an application executing in a containerized environment, according to various embodiments.

FIG. **8** is an example of a process for detecting network connections, detecting components within a containerized environment, and generating a network policy for a component of an application executing in the containerized environment, according to various embodiments.

FIG. **9** is an example of a process of the manner in which the network policy of a component may be utilized, according to various embodiments.

FIG. **10** is a block diagram illustrating one pattern for implementing a cloud infrastructure as a service system, according to at least one embodiment.

FIG. **11** is a block diagram illustrating another pattern for implementing a cloud infrastructure as a service system, according to at least one embodiment.

FIG. **12** is a block diagram illustrating another pattern for implementing a cloud infrastructure as a service system, according to at least one embodiment.

FIG. **13** is a block diagram illustrating another pattern for implementing a cloud infrastructure as a service system, according to at least one embodiment.

FIG. **14** is a block diagram illustrating an example computer system, according to at least one embodiment.

DETAILED DESCRIPTION

In the following description, for the purposes of explanation, specific details are set forth in order to provide a thorough understanding of certain embodiments. However, it will be apparent that various embodiments may be practiced without these specific details. The figures and description are not intended to be restrictive. The word “exemplary” is used herein to mean “serving as an example, instance, or illustration.” Any embodiment or design described herein as “exemplary” is not necessarily to be construed as preferred or advantageous over other embodiments or designs.

In certain approaches, software defined networks, used in container orchestration tools often by default, allow communication between all the components within the system. For example, in Kubernetes, Common Network Interface plugins are required to assign an Internet Protocol (IP) address to every component (or pod) in an application and the plugins allow traffic between every pod, regardless of the underlying network used. One approach involves the use of software-based tunneling protocols, such as VXLAN, to provide a flat network that moves traffic seamlessly within and between hosts (physical machines or virtual machines) that provide computing instances to the cluster. While, the primary goal of such a network is to provide connectivity for continuously changing workloads, it does not provide security or isolation of network traffic between the components.

To compensate for the challenges faced by container orchestration tools that use traditional network controls to specify policy rules for communication between the components of a containerized application, container orchestration frameworks may provide mechanisms to restrict the traffic within the cluster by specifying a network policy (e.g., a network security policy) configuration for the entire application and providing it to the cluster’s control plane. The policy indirectly addresses the components through metadata associated with these components. For example, in Kubernetes, the policy specification may not use identifiers of pods, but may instead use pod labels and namespaces where multiple pods can share labels and namespaces. As used herein, a pod may refer to a set of one or more components of an application that may be processed by the orchestration tool at a time.

In certain approaches, in container-based frameworks such as Kubernetes and OpenShift, network policies are distributed as a collection of independent objects. These objects have their own lifecycle, can be changed independently, but affect each other and the system (i.e., the containerized application) as a whole. For example, a policy statement for one object can accidentally overshadow another policy defined for another object. Furthermore, elements that are used to specify the policy, such as component labels, may change independently from the policy itself. It is possible that routine reconfiguration of a component will accidentally impact a policy. Additionally, when tasks of the component and network policy management are split between different teams within an organization, the process of defining network policies for individual components by container-based frameworks becomes even more challenging.

Network policy management is particularly challenging with a truly continuous delivery model. In this case, individual components are delivered independently, typically with one change to the system at a time. New components may be delivered very frequently, multiple times a day, and each time they may potentially require a different network configuration. In addition, older and newer versions of the same component are often running in in parallel, to facilitate non-disruptive transition during an upgrade, or to provide a gradual change delivery model. In this model, system processing (defined by a portion of processing, groups of users or other means) is gradually directed to newer components. Gradual rollout of changes allows monitoring changes in a real-life environment and their operation under increased load, without risking a full system disruption. Also, the arrangement of the components may be different in different environments, with different feature sets, regulatory compliance requirements or customer needs.

In such an environment, defining precise network policies is a challenging task. The requirements for policies and connection paths may change on daily basis and manual creation of tight network policies is not feasible at this rate, speed and degree of automation. There is no easy way to infer network policies from the observed traffic. When the traffic is recorded, the only information directly available is low level (IP addresses, ports, protocols, etc.). This information however cannot be used to construct the policies, and are not stable or repeatable across clusters or even within single cluster over time. Also, the elements that are used to specify the policies, such as component labels, may change independently from the policies themselves. It is possible that routine reconfiguration of a component will accidentally impact a policy. Tasks of component and network policy management can be split between different parts of the team which makes this problem even more likely. In practice, many organizations resort to using a pre-defined set of policies (also referred to herein as coarse-grained network policies) such as the zone-based polices to reflect general network traffic expectations of the system. While a coarse-grained policy developed by users may be used to reflect general network traffic expectations of the system, in container frameworks it is not possible to use a coarse-grained together with a fine-grained policy. This is due to the fact that, if present, the coarse-grained policy would allow all the traffic within its boundaries and overshadow its subset, that is, the fine-grained policy.

The present disclosure describes improved techniques for automatically inferring software-defined network policies from the observed workload in a computing environment. The disclosed systems include capabilities for capturing the traffic data in the cluster and simultaneously correlating the source and destination addresses with the objects in the cluster. This provides the data on communication between the objects in the cluster. This data is subsequently aggregated and used to create the network policies. In operation each component/node on a network is provisioned to include an agent that monitors the outgoing traffic. The agent records network flow (e.g., TCP connections and UDP traffic) within the cluster. Once a flow is recorded, the components corresponding to the source of destination addresses are identified by the agent and recorded with the flow. Identification and recording of components simultaneously with the traffic is advantageous, as during the cluster lifecycle components may be added or removed, addresses reused and so forth. Along with the component, the agent records the component metadata that is relevant for creating a policy such as component labels. This data, including the specific network flow between two cluster objects is recorded. Simultaneously, the agent records the status and changes to all of the objects in the cluster. Understanding of all the objects is advantageous as it allows to identify the components that are not involved in any traffic flows. By the end of the process, all the cluster components, as well as their communication is recorded. This data is subsequently used to generate a network policy.

Referring now to the drawings, FIG. **1** depicts an example of a distributed computing environment **100** for generating network policies such as network security policies for components of an application deployed in the computing environment, according to certain embodiments. The computing environment **100** may include a test environment **112** and a production environment **122**. The test environment **112** and the production environment **122** may comprise one or more computing systems that execute computer-readable instructions (e.g., code, program) to implement the test environment **112** and the production environment **122**. As depicted in FIG. **1**, the test environment **112** includes a test system **108** and the production environment **122** includes a deployment orchestrator system **116**. Portions of data or information used by or generated in the test environment **112** and the production environment **122** as part of its processing may be stored in a persistent memory such as a network policy data store **120**. The systems depicted in FIG. **1** may be implemented using software (e.g., code, instructions, program) executed by one or more processing units (e.g., processors, cores) of a computing system, hardware, or combinations thereof. The software may be stored on a non-transitory storage medium (e.g., on a memory device).

The computing environment **100** depicted in FIG. **1** is merely an example and is not intended to unduly limit the scope of claimed embodiments. One of ordinary skill in the art would recognize many possible variations, alternatives, and modifications. For example, in some implementations, the computing environment **100** can be implemented using more or fewer systems than those shown in FIG. **1**, may combine two or more systems, or may have a different configuration or arrangement of systems and subsystems.

The computing environment **100** may be implemented in various different configurations. In certain embodiments, the computing environment **100** comprising the test system **108** and the deployment orchestrator system **116** may be implemented in an enterprise servicing users of the enterprise. In other embodiments, the systems in the computing environment **100** may be implemented on one or more servers of a cloud provider and the network policy creation services of the systems may be provided to subscribers of cloud services on a subscription basis.

In certain embodiments, a user may interact with the test system **108** using a user device **102** that is communicatively coupled to the test system **108**, possibly via one or more communication networks. The user device **102** may be of various types, including but not limited to, a mobile phone, a tablet, a desktop computer, and the like. The user may represent a user of an enterprise who subscribes to the services provided by the systems of the computing environment **100** for automatically generating network policies for components of an application to be deployed in the computing environment. The user may interact with the test system **108** using a browser executed by the user device **102**. For example, the user may use a user interface (UI) (which may be a graphical user interface (GUI)) of the browser executed by the user device **102** to interact with the test system **108**.

In certain embodiments, a user may, via the UI, provide an application **104** to be deployed in the computing environment. The application **104** may represent a micro-service based containerized application that may be deployed in the production environment **122**. In certain examples, the application **104** may comprise multiple components where multiple instances of each component can be executed as containers on nodes within a cluster of nodes in a containerized environment of the production environment **122**. In certain examples, the containerized environment may be provided by a container orchestration platform such as Kubernetes, OpenShift, Docker Swarm and the like. An example of an application deployed to a production environment **122** is shown in FIG. **2**.

In certain examples, the application **104** (comprising a set of one or more components) may be provided to the test system **108** prior to its deployment in the containerized environment. As part of the testing phase, the test system **108** monitors the network traffic flow between the different components of the application and generates component-specific network policies **110** for the components based on the discovered network paths/network traffic. In certain embodiments, for each network path identified for a component, the test subsystem **108** performs additional processing to check if the network path violates a pre-defined network policy **117** (also referred to herein as a coarse-grained network policy) defined for the test environment/application. The set of pre-defined network polices **117** may be defined, for example, by the user of the enterprise and these policies reflect general network traffic expectations of the flow of network traffic between the components of the application in the test environment. Examples of coarse-grained polices defined for an application are described in FIG. **4**. If it is determined that the coarse-grained network policy is not violated by any network path identified for the component, the test system **108** generates a component-specific network policy for the component.

In certain embodiments, as part of the application development process, the test subsystem **108** may be configured to receive (via the UI) a new version of a component **106** of the application **104** to be deployed as part of the application deployed in the containerized environment. For instance, a user of an enterprise may wish to update an earlier version of a component of the application, for example, because the operation performed by the component changed. In this case, the user may provide an updated (or new) version of the component for testing by the test system **108**. The test subsystem **108** receives the updated version of the component **106** and generates a component-specific network policy for the updated version of the component by identifying network paths/network traffic originating from the updated version of the component to one or more other components within the application. Additionally, the test subsystem **108** performs processing to determine if any of the identified network paths violate the coarse-grained network policy defined in the test environment.

In certain embodiments, as a result of the processing performed by the test system **108**, the test system **108**, generates a deployment package **114** that includes the component(s) of the application **104** to be deployed and their associated network polices. A deployment orchestrator system **116** in the production environment **122** receives the deployment package **114** and uses the deployment package to deploy the component(s) of the application and their associated network policies to different nodes in a cluster of nodes in the containerized environment. In certain examples, the deployment orchestrator system **116** stores information identifying the network policies associated with the different components in the network policy data store **120**.

In certain situations, to facilitate non-disruptive transition during an upgrade, or to provide a gradual change delivery model during the application development process, both an earlier version of the component and an updated (or new) version of the component may need to co-exist and execute in parallel the containerized environment for some time. In certain embodiments, the test system **108** and the deployment orchestrator system **116** include capabilities for enabling different versions of a component of a containerized application to co-exist on different computing nodes in a cluster of nodes of the containerized environment at the same time. The systems additionally include capabilities for enabling different network policies to be generated for and applied to the different versions of the component, where each component has potentially different network requirements. Additional details related to the processing performed by the various systems of the computing environment **100** in FIG. **1** are described below in FIGS. **2-7**.

FIG. **2** depicts an example of an application deployed in a containerized environment of the computing environment **100** shown in FIG. **1**, according to certain embodiments. In the depicted example, the application comprises an order processing application **200** that is deployed in a container-based framework within the production environment **122** of the computing environment **100**. Multiple instances of each component are executed as containers on nodes within a cluster of nodes within the container-based framework. By way of example, the container-based framework may be implemented using a container orchestration tool such as Kubernetes, OpenShift, Docker Swarm and the like.

In certain examples, the order processing application **200** may include a set of components. These components may include, but are not limited to:

- a static web application front end component **202** providing user experience in the web browser;
- an Application programing Interface (API) gateway component **204** to the application, responsible for processing API calls from the web application component **202** and forwarding them to respective services implementing the business logic;
- a collection of service components providing distinct functions of the application such as processing orders, invoices and user management. By way of example, the collection of service components may include, but are not limited to, an order-service component **206**, a user-service component **208** and an invoice-service component **210**; and
- data middleware component for storing and processing data, including a database component **212** and a message queue component **214**.

In certain examples, for purposes of discussion, it is assumed that the components **202-214** can communicate directly with one another. In a certain implementation, each component is denoted by its name and a corresponding zone label. By way of example, the web application front end component **202** may be denoted by a name "web-app" and a corresponding zone label "web." Additional details of how zone labels are generated for components of a containerized application are described in FIG. **4**. Further, in the depicted example, the web application front end component **202** does not communicate with the other components that make up the application **200** but the API gateway component **204** communicates with the services **206**, **208** and **210**. Additionally, in this example, all the service components, **206**, **208** and **210** communicate with each other (but not necessarily each service reaches each other service), the services communicate with the database service **212** and the order-service **206** and invoice-service **210** access the message queue **214**.

The different components of the order processing application 200 depicted in FIG. 2 are merely an example and is not intended to unduly limit the scope of claimed embodiments. One of ordinary skill in the art would recognize many possible variations, alternatives, and modifications. For example, in some implementations, the order processing application 200 may comprise more or fewer components than those shown in FIG. 1, may combine two or more components, or may have a different configuration or arrangement of components.

FIG. 3 depicts an example arrangement of components of the application shown in FIG. 2 on a cluster of nodes 300 in a container-based framework, according to certain embodiments. Each node in the cluster of nodes is assigned an Internet Protocol (IP) address. In the depicted example, the cluster of nodes 300 consists of four nodes (node 1, node 2, node 3 and node 4), each with an IP address within the range of its network, such as cloud Virtual Cloud Network (VCN) or physical network. The containers, residing on each of the nodes in the cluster use a different IP address range. Traffic between components on different nodes is typically tunneled and container-to-container packages are encapsulated as node-to-node packages. The depicted example illustrates a specific arrangement of the components in a point in time snapshot of the container-based framework. The framework may re-arrange the components or move them between nodes to distribute the load. Additionally, the specific containers may fail and new instances may get created, nodes may be added or removed and so forth. In each such case, since the IP addresses of the containers that change will also likely change, any network filtering controls (e.g., network security policies) that are based on identifying components using their IP addresses may be impractical or infeasible to implement by the container-based framework.

As previously described, in order to compensate for these challenges, container orchestration frameworks may include capabilities for controlling the communication (i.e., network traffic) between the components of the application by using a set of one or more pre-defined (coarse-grained) network policies. For example, in a container-based framework such as Kubernetes or OpenShift, the network policy may be defined based on metadata of the specific components such as “zone labels” as shown in FIG. 4. Additional examples of network policies defined by container-based frameworks for controlling network traffic between multiple components of a containerized application are described in FIG. 4.

FIG. 4 depicts various examples of network polices defined by a container-based framework for a containerized application deployed on a cluster of nodes in the container-based framework, according to certain embodiments. For purposes of discussion, the network policies 402, 404 and 406 are described in relation to the order processing application 200 described in FIGS. 2 and 3. In one implementation, as shown in table 402, the set of policies may be defined using a “zone label” as a source label and a destination label for the components of the containerized application (e.g., 200). In a certain implementation, the set of policies may be implemented using JavaScript Object Notation (JSON) objects or as YAML objects. The zone-based network policies depicted in table 402 may permit some traffic that is not part of the regular system operation, such as connectivity from the user-service component 208 to the database component 212. These zone-based policies will not require to be updated when routine incremental changes occur to the containerized application, such as when more services or database middleware are added to the application. The zone-based policies also continue to be operational if more services begin to communicate with each other, such as, for example, the order-service component 206 and the invoice-service component 210. Also, the zone-based policies defined by the container-based framework typically allow traffic between the components, unless a certain path is specifically covered by the policy. For this reason, in order to deny the internal traffic to certain components (such as ‘web’ or ‘front’ zones), the zone-based polices may include policy directives with an empty origin denoted with ‘-’.

In another approach, as shown in table 404, the set of policies may be implemented using an “app label” as a source label and a destination label where the “app label” uniquely identifies the components of the container-based application. In this implementation, the policy that most precisely reflects the network traffic in the system will take the form shown in table 404. Such a policy may disallow any traffic not expected in the system but changes to the application will require corresponding policy changes to be applied to the components of the application.

In yet another implementation, the container-based framework may implement a more balanced policy that takes advantage of both of the “zone label” and the “app label” used to uniquely identify the components of the application. For example, a security analysis of the application may reveal that a key component of concern is the message queue component 214. While it is acceptable to have general zone-based rules between the api-gateway component 204 and the services 206, 208 and 210, the services themselves, the database component 212 and the communication with the message queue component 214 need to be strictly restricted to the services that need the access. In such a case, the set of policies may take a form shown in table 406. It is to be appreciated that the terms, “zone label” and “app label” used in this disclosure are illustrations of one type of grouping of components that may be utilized in an application for defining the flow of network traffic within the application. For instance, a “zone label” may identify a first group of components to which a first version of the component can send traffic and an “app label” may identify a second group of components to which the first version of the component can send traffic. The “zone label” may identify a larger group (e.g., a set) of components that the first version of the component can send traffic while the “app label” may identify a specific set (e.g., a subset) of components within the larger group that the first version of the component can send traffic. In alternate embodiments, different label names, different label groups or different layers of labels (e.g., a zone cell, a four layer model for referencing the component and the like) may be used to identify and group components of an application.

As previously described, in certain situations, a user of an enterprise may wish to add an updated version of an existing component of the application when the operation performed by an earlier/previous version of the component changes. For instance, based on the example of the containerized application 200 shown in FIG. 2, an updated version of the user-service component 208 may be required to communicate with the message queue component 214 instead of the database component 212. FIG. 5 is an example of the containerized application shown in FIG. 2 depicting a change in the operation of one of the components within the containerized application, according to certain embodiments. In the embodiment depicted in FIG. 5, a change in the operation performed by the user-service component 208 results in the user-service component 208 communicating with the message queue component 214 instead of the database component 212. If the change in the operation has to be gradually enabled in the application, both the previous version as well as the updated version of the user-service component **208** have to co-exist and operate in the deployed application for some time. In this situation, the deployed application **200** will contain two versions of the user-service component **208**, a previous version that requires network access to the database component **212**, but not to the message queue component **214**, and another (updated) version that requires access to message queue component **214**, but not to the database component **212**.

A coarse-grained policy, based on "zone labels" as depicted in table **402** in FIG. **4** may allow the containerized application to operate/execute correctly with either version of the component and with both versions of the components running together. However, such a coarse-grained policy may not be considered sufficiently secure as it allows communication that is not part of regular system operation. At the same time, the user (e.g., a system administrator) would prefer that any policy would be contained in the boundaries defined by the coarse-grained policy.

A fine-grained policy, for example, based on "app labels" as depicted in table **404**, will result in a containerized application that initially allows traffic from the user-service component **208** to the database component **212** only. This policy will prevent the updated version of the user-service component **208** to communicate with the message-queue component **214** after the updated version is deployed. However, an updated policy that only allows communication with the message-queue component **214** will disrupt the previous version. Finally, a single policy which allows communication with both versions of components in the data zone may not be sufficiently precise either. Such a policy will allow the previous version of the user-service component to communicate with the message-queue component **214** and the updated version of the user-service component to communicate with the database component. However, both those paths may be overly permissive.

To address these challenges, the disclosed system provides improved techniques for automatically inferring fine-grained network policies (e.g., network security policies) from the observed workload in the computing environment. The system provides a mechanism to create precise, per-component network policies, while respecting any coarse-grained policies of the containerized application and/or prior existing fine-grained network policies. In certain embodiments, the system includes capabilities for aligning the policy management with the lifecycle of corresponding components of the containerized application. The system builds network policies that are linked to particular deployment packages. The techniques for generating per-component policies by the disclosed system is seamless and does not require additional labelling by users of the enterprise. Additional details of the operations performed by the systems shown in FIG. **1** for generating network policies for different components are described in FIG. **6**.

FIG. **6** depicts additional details of the operations performed by the systems shown in FIG. **1** for generating network policies for components of an application deployed in a computing environment, according to certain embodiments. The computing environment **600** may be implemented by one or more computing systems that execute computer-readable instructions (e.g., code, program) to implement the computing environment. As depicted in FIG. **6**, the computing environment **600** may include a test environment **112** or a production environment **122** as described with respect to FIG. **1**. The test environment or production environment **112/122** includes one or more nodes **605** (*a-n*) that host one or more containerized components **610** (*a-n*). The test environment or production environment **112/122** assigns a specific IP address or port range to the components **610** and provides an ability (e.g., an application programming interface (API)) to discover the components **610** through the IP address/port. A network monitoring agent **615** is provisioned to run on the nodes **605** and adapted to monitor traffic originating from the components **610** through the IP address/port. Depending on the container framework, and the network implementation, the network monitoring may be implemented as part of every component (container) or as a single instance for the node. The decision on the location may be based on the practical reasons and availability of instrumentation mechanisms within the specific container framework. In some configurations, such as when the container network is used without an overlay/tunneling, but instead built on top of the lower-level network, the component may be centrally located within the network. In addition, existing cluster monitoring tools, such as Prometheus for Kubernetes, could be used to supply the data to the system.

The test environment or production environment **112/122** further includes a component lookup module **620**, a component monitor module **625**, a policy creator **630**, a connections queue data store **635**, and a traffic data store **640**. The component lookup module **620** is adapted to identify the components **610** based on their IP address/port. The component monitor module **625** is adapted to actively monitor the cluster of nodes **605** for new components **610** being added to the nodes **605** or component **610** changes within the nodes **605**. The policy creator **630** is adapted to create network policies from the traffic data monitored by the network monitoring agents **615**. The connections queue data store **635** is a queue of traffic data supplied by the network monitoring agents **615** for processing by the component lookup module **620**, component monitor module **625**, and policy creator **630**. The traffic data store **640** is a set of component-to-component flow data that includes connections between components **610** monitored/identified by the network monitoring agents **615**, the component lookup module **620**, and the component monitor module **625**. The test environment or production environment **112/122** is adapted to allow for the specification of network policies **655** (via the policy creator **630**) that are based on component metadata for components **610** discovered/identified by the component lookup module **620** and the component monitor module **625**. The test environment or production environment **112/122** may receive commands and input from a container framework control plane **645** and/or an operator **650**. Portions of data or information including the network policies **655** used by or generated by systems of the computing environment **600** as part of its processing may be stored in a persistent memory such as a network policy data store **660**.

The systems and subsystems depicted in FIG. **6** may be implemented using software (e.g., code, instructions, program) executed by one or more processing units (e.g., processors, cores) of a computing system, hardware, or combinations thereof. The software may be stored on a non-transitory storage medium (e.g., on a memory device). The computing environment **600** depicted in FIG. **6** is merely an example and is not intended to unduly limit the scope of claimed embodiments. One of ordinary skill in the art would recognize many possible variations, alternatives, and modifications. For example, in some implementations, the computing environment **600** can be implemented using more or fewer subsystems than those shown in FIG. **6**, may combine two or more subsystems, or may have a different configuration or arrangement of subsystems.

In certain embodiments, a user (e.g., via the UI of a user device **102**) may provide a component **610** of an application (e.g., **104**) to the test environment or production environment **112**/**122** to be deployed as part of the application deployed in the computing environment. For instance, as part of the application development process, the user may develop a new component for the application or may create an updated version of an existing component of the application for deployment in the computing environment. As previously described, the application may represent a containerized application deployed in the production environment **122**. In certain examples, the application may comprise multiple components **610** where multiple instances of each component may be executed as containers on nodes **605** within a cluster of nodes in the production environment **122**.

In various embodiments, an acceptance test subsystem within the test environment **112** receives the component **610** and tests it prior to its deployment in the containerized environment. In certain embodiments, testing the component may involve generating network policies regarding the components, performing a policy compliance test, and performing a functional acceptance test. The generating the network policies and performing a policy compliance test for the component, in certain examples, involves, obtaining, by the acceptance test subsystem, information identifying the network connections (i.e., the network paths) originating from the various components **610** on the nodes **605**. In certain embodiments, the network paths originating from the component **610** may be discovered by the network monitoring agent **615** that is communicatively coupled to the acceptance test subsystem via a communication network. The network monitoring agent **615** monitors the traffic originating from the network interfaces corresponding to the containers that execute the application's components **610**. Once a new network connection is detected (e.g., upon integration of the new component being tested into a node), the network monitoring agent **615** records details of the new network connection data (e.g., source and destination address, ports, etc.) and the time of the connection (e.g., a time stamp), which is sent to the connections queue data store **635** for downstream processing.

While the network monitoring agent **615** records details of the new network connection, the component monitor module **625** observes the components **610** through the framework's control plane **645**. The component monitor module **625** records any changes to component arrangement (e.g., integration of the new component being tested into a node), corresponding network addresses, and time of changes, and stores the data in a cache or local data store **665** for lookup. Along with the corresponding network address and time, the component monitor module **625** also obtains, through the framework's control plane **645**, component metadata (e.g., labels for the components **610**) that is relevant to the creation of network policies, and stores the component metadata in association with the data (e.g., component arrangement, corresponding network addresses, and time of changes) in the cache or local data store **665** for lookup. The component lookup module **620** processes the new network connection data from the connections queue data store **635**. For the new network connection data in the connections queue data store **635**, the component lookup module **620** communicates with the component monitor module **625** providing details of each new network connection data (e.g., source and destination address, ports, time stamp, etc.), and receives the information about the components **610** involved in the new network connection (e.g., component arrangement and network addresses) and their metadata, matching corresponding network addresses and time. The component lookup module **620** then stores the processed information (component metadata for components at both ends of new network connection, network addresses, ports, time stamps, etc.) in the traffic data store **640**. The traffic data store **640** has a form of a set, and any duplicate information will not be recorded.

In order to identify cluster components **610** from the network connection data, the addresses for the traffic need to be matched to the corresponding components. This may be implemented in multiple ways. In some instances, this may be implemented by actively querying the container framework control plane **645** every time a connection is observed, by the component monitor module **625**. Such approach results in simple implementation but increases the overhead of the discovery process, and may not be feasible in configurations with large amounts of network connections. The network monitoring agent **615**, located close to the source, may already contain enough information about the source of the network connection to provide that data. Then, only the destination information requires a query. In other instances, the component monitor module **625** caches the information about IP/port to component mapping. This information may be reused to reduce the number of queries. An extension of this approach is to have the component monitor module **625** monitor the cluster changes and keep the current mapping available for fast lookup. Finally, the component monitor module **625** can retain entire history of the cluster components along with their addresses and the time of any changes. This way every connection can be mapped to the corresponding component at a later stage. This may allow the correlation operation to be done asynchronously, or even as a batch operation after the data collection is completed. The system may provide an insight into the process by reporting how many new communication or data paths are found. If no new path is found and the system is known to operate with full set of features, this may indicate that sufficient data has been gathered.

After obtaining the processed information identifying the network connections originating from the components **610**, as part of performing the policy compliance test for a given component (e.g., the new or updated component), the acceptance test subsystem then determines if the network connections that the component are attempting to use are in compliance with the coarse-grained policy defined for the application. As noted herein, a coarse-grained policy may be developed by a user (e.g. a developer of the enterprise) of the application. For instance, a coarse-grained policy for an application (e.g., **114**) shown in FIG. **2** may be developed using "zone labels" as shown in table **404** of FIG. **4**.

In certain embodiments, in addition to the policy compliance test, the acceptance test subsystem may also perform functional testing of the component to confirm that the functionality of the component is behaving as expected. The functional testing may be performed before, after, or in parallel with generating policies and performing the policy compliance test for the component. As a result of performing the functional acceptance test and the policy compliance test for the component, the acceptance test subsystem outputs a test result. The test result outputs information indicative of the success or failure of the functional test and/or the policy compliance test. For instance, if the component did not successfully pass the functional test (in presence of the coarse-grained network policy), the test result may output information indicating that the functional test failed and that the component cannot not be deployed in the containerized environment. The information that is output by the test result in this case may also indicate that a functional bug exists in the execution of the component. If the failure was a result of a coarse-grained policy violation, the test result may output information indicating that the component is attempting to use a network path that is not allowed by the coarse-grained policy. If the component passes the functional test, but a coarse-grained policy violation was detected, the test result may output information indicating that the component will not deployed due to a problem with test coverage.

If the component passes the functional test and no violation of the coarse-grained policy is detected, the test result may indicate that the component can be successfully deployed to production. In this case, the policy creator **630** accesses the traffic data store **640**, retrieves the pertinent processed information (component metadata for components at both ends of new network connection, network addresses, ports, time stamps, etc.), and creates a network policy **655** (i.e., a fine-grained component-specific network policy) for the component. Various different techniques may be used to create a network policy. An example of one such technique is described in U.S. patent application Ser. No. 17/124,162, entitled "Techniques For Generating Network Security Policies For Application Components Deployed In A Computing Environment," and filed concurrently with the present application. The entire contents of the aforementioned application is incorporated herein by reference in its entirety for all purposes. Depending on the nature of the network policy **655** in the specific application framework, the policy creator **630** may also contact the component monitor module **625** to obtain a list of components in the system to identify ones that did not take part in any connection. This may be required to explicitly block the traffic to those components via the network policy **655**. The policy creator **630** may also take additional configuration input from the system operator **650**. The input from the system operator **650** may inform how the network policy **655** is to be created, for example what properties of the components should be used, whether some components should be excluded from the network policy **655**, and so forth. The network policy **655** is then created.

In some embodiments, creating the network policy **655** for the component involves, generating, by the policy creator **630**, a component identifier for the component and associating the component with the component identifier. In certain examples, the component identifier is a "version label" that identifies the current version of the component. The version label for a component may represent, for instance, a current timestamp, a commit identifier, or a version identifier for the component. The policy creator **630** uses the version label and the portion of the network traffic (from the network paths discovered by the monitoring agent **615**) originating from the currently processed component to generate a fine-grained network policy **655** for the component. Examples of fine-grained policies generated by the policy creator **630** are described in detail herein, using the example of the order processing application depicted in FIG. **2**. The component, fine-grained policy and other deployment artifacts (such as deployment manifests) are packaged together to create a deployment package for the component.

In certain examples, a deployment orchestrator system in the production environment **122** receives the deployment package and uses the deployment package to deploy the component(s) of the application and their associated network policies to different nodes in a cluster of nodes within a containerized environment of the production environment **122**. The deployed application executes in the containerized environment. In a specific implementation, the containerized environment may be provided by a container orchestration tool such as Kubernetes or OpenShift. In certain examples, the deployment orchestrator system may store the fine grained network policies **610** associated with the component(s) in the network policy data store **660**.

In certain embodiments, the deployment orchestrator system may be configured to align the lifecycle of a component with its corresponding fine-grained network policy. For example, for a container-based orchestration platform such as Kubernetes or OpenShift, the alignment of the lifecycle of the component with its corresponding fine-grained network policy can be implemented using an Operator pattern that replaces the regular interfaces related to deployment of the component and instead accepts a deployment package comprised of the component and its associated fine-grained network policy. In other examples, an admission controller may be used to capture any operations on the pod. Such a controller may automatically add or remove a corresponding network policy with a matching label when the component is added/removed from the cluster. A network policy and its component may also be packaged together in a higher-level object such as a Helm chart that will automatically control the deployment of multiple objects together.

In various embodiments, the operations performed by the systems shown in the computerized environment **600** described above may be combined with a continuous integration/continuous deployment process within the production environment **122**. The process may cover the entire lifecycle from building a component to integration/acceptance testing of the component and deployment of the component in the cluster of nodes. In some embodiments, the data capture (e.g., connection and component discovery) may be done initially in the production environment without active policy or with course-grained active policies. After sufficient data is gathered, the fine-grained network policies may be created and enabled. As discussed with respect to the testing environment **112**, the network paths originating from the components **610** may be discovered by the network monitoring agent **615** during deployment. The network monitoring agent **615** may continuously monitor the traffic originating from the network interfaces corresponding to the containers that execute the application's components **610**. Once a network connection is detected (e.g., upon integration of the new component being tested into a node) or a change in a network connection is discovered, the network monitoring agent **615** records details of the new or changed network connection data (e.g., source and destination address, ports, etc.) and the time of the connection (e.g., a time stamp), which is sent to the connections queue data store **635** for downstream processing.

While the network monitoring agent **615** records details of the network connections in deployment, the component monitor module **625** observes the components **610** through the framework's control plane **645**. The component monitor module **625** records any changes to component arrangement (e.g., replacement of components that fail during deployment), corresponding network addresses, and time of changes, and stores the data in a cache or local data store **665** for lookup. Along with the corresponding network address and time, the component monitor module **625** also obtains, through the framework's control plane **645**, component metadata (e.g., labels for the components **610**) that is relevant to the creation of network policies, and stores the component metadata in association with the data (e.g., component arrangement, corresponding network addresses, and time of changes) in the cache or local data store **665** for lookup. The component lookup module **620** processes the network connection data from the connections queue data store **635**. For the network connection data in the connections queue data store **635**, the component lookup module **620** communicates with the component monitor module **625** providing details of each network connection data (e.g., source and destination address, ports, time stamp, etc.), and receives the information about the components **610** involved in the network connection (e.g., component arrangement and network addresses) and their metadata, matching corresponding network addresses and time. The component lookup module **620** then stores the processed information (component metadata for components at both ends of new network connection, network addresses, ports, time stamps, etc.) in the traffic data store **640**.

The policy creator **630** accesses the traffic data store **640**, retrieves the pertinent processed information (component metadata for components at both ends of new network connection, network addresses, ports, time stamps, etc.), and dynamically creates network policies **655** (i.e., a fine-grained component-specific network policy) for the components based on changing conditions within the nodes and component arrangements. Depending on the nature of the network policy **655** in the specific application framework, the policy creator **630** may also contact the component monitor module **625** to obtain a list of components in the system to identify ones that did not take part in any connection. This may be required to explicitly block the traffic to those components via the network policy **655**. The policy creator **630** may also take additional configuration input from the system operator **650**. The input from the system operator **650** may inform how the network policy **655** is to be created, for example what properties of the components should be used, whether some components should be excluded from the network policy **655**, and so forth. The network policy **655** is then created as described with respect to the test environment **112**.

Additional details related to the operations performed by the systems shown in the computerized environment **600** for generating component specific network policies (i.e., fine-grained network policies) for components of an application deployed in a containerized environment are now described using the order processing application **200** depicted in FIGS. **2** and **5**. In a specific implementation, the containerized environment is provided by a container orchestration platform such as Kubernetes. The cluster of nodes is initially configured with a deny-all policy. Such a policy can be created by matching a component (or a pod) with a label (e.g., a zone label or an app label depicted in FIG. **4**) while specifying no network traffic. As previously described, a pod may refer to a set of one or more components of an application that may be processed by the orchestration tool at a time.

A coarse-grained policy for the order processing application **200** may be represented using "zone labels" as shown in table **402**. This may be used in the test environment of the test system to perform a policy compliance test for the component as described in FIG. **6**. Now consider that the test system receives an update to the user-service component **208** or a change occurs to the user-service component **205** during deployment. Before receiving the new version of the user-service component **205** or replacing a user-service component **208** in deployment, the component-specific network policy (i.e., the fine-grained network policy) related to user-service component **208** may be depicted as shown below:

{app: user-service, zone: services, version: 1582630603}→{app: database}

The fine-grained policy reflects a current, and at this point in time, the only version of the user-service component **208** that is granted access to the database component **212**. A user of the system prepares an updated version of the user-service component **208** or the system itself identifies a defective user-service component **208** in deployment and prepares another version of the user-service component **208** to be placed online. This version of the user-service component **208** no longer uses the database component **212** but uses the message-queue component **214** instead. It is intended that the new or updated version of the user-service component **208** will co-exist with the previous version for some time to allow a smooth transition in the operation of the application. Using the zone labels and the app labels shown in FIG. **4**, the user-service component **208** may be identified using the following labels:

{app: user-service, zone: services}

When the updated version of the user-service component **208** is received, the component is packaged in a container and deployed into the test or production environment. The test or production environment is implemented as a Kubernetes workload and the user-service component **208** is deployed in the cluster. Component lookup module **620** and component monitor module **625**, policy creator **630** and data stores **635**; **640**; **660**; **665** may be located in a separate namespace in test or production environment. In addition, the monitoring agents **615** are deployed on every worker nodes using for example a DeamonSet that will guarantee a single instance running on every node. The monitoring agents **615** run as privileged containers and set up local firewall (iptables) rules to receive the traffic from the virtual interfaces corresponding to the pods in the system. The component monitor module **625** may be operating with a service account that provides a read access to the Kubernetes API server and sets up a webhook that makes API server to send all the changes as they occur within the test or production environment. The components are configured to ignore the network traffic corresponding to the system itself, such as traffic originating from monitoring agents **615**. During acceptance testing of the component or during instantiation of the component in the production environment, a network path from the user-service component **208** to the message-queue component **214** is discovered by the monitoring agent **615**.

For example, as the application runs it may be subjected to the test automation, the traffic data is gathered by monitoring agent **615**, which is recorded to the connections queue **635**. For example, following the arrangement in FIG. **3**, the initial data captured may be as follows:

. . .
10.244.1.3→10.244.1.5:443
10.244.2.8→10.244.2.12:443
10.244.1.3→10.244.2.9:443
10.44.1.4→10.244.1.6:443
10.44.1.4→10.244.4.22:443
. . .

Note, that for simpler presentation the source ports have been omitted that are not relevant in this specific example. The timestamps have also been omitted as, in the scope of this example it is assumed that the arrangements of pods will not change.

In the meantime, the component monitor module **625**, builds a list of components **610** in the system, reflecting the arrangement in FIG. **3**.

. . .

10.244.1.3→{name: invoice-service, labels: {zone: services, app: invoice-service}}

10.244.2.8→{name: invoice-service, labels: {zone: services, app: invoice-service}}

10.244.1.4→{name: order-service, labels: {zone: services, app: order-service}}

10.244.1.6→{name: database, labels: {zone: data, app: database}}

10.244.2.12→{name: database, labels: {zone: data, app: database}}

10.244.2.9→{name: message-queue, labels: {zone: data, app: message-queue}}

10.244.4.22→{name: message-queue, labels: {zone: data, app: message-queue}}

. . .

The component lookup module **620** processes the network connection information, that results in:

. . .

{name: invoice-service, labels: {zone: services, app: invoice-service}}→{name: database, labels: {zone: data, app:
database}, port: 443} {name: invoice-service, labels: {zone: services, app: invoice-service}}→{name: message-queue, labels: {zone: data, app:
message-queue}, port: 443} {name: order-service, labels: {zone: services, app: order-service}}→{name: database, labels: {zone: data, app:
database}, port: 443} {name: order-service, labels: {zone: services, app: order-service}}→{name: message-queue, labels: {zone: data, app:
message-queue}, port: 443}

. . .

Note, the traffic captured in this example, covers the initial connections from order and invoice services to the database and message queue. Even though the 5 connections are analyzed, the captured traffic information results in only 4 items, as the first and second connections are two different instances of the same invoice service component and result in the same traffic data. The process continues through the entire lifecycle of the test phase.

The test or production environment may include a coarse-grained network policy which may be represented as shown below:

{zone: services}→{zone: data}

This coarse-grained network policy allows general traffic between services and data zones. The acceptance test passes successfully and there is no policy violation detected. The policy creator **630** creates a "version label" for the user-service component **208** and associates a "version label" identifier with the component. As an example, the "version label" identifier may be set to '1582650759' (which, in this example, represents a current timestamp). As a result, the updated version of the user-service component **208** may be identified using the labels shown below:

{app: user-service, zone: services, version: 1582650759}

The policy creator **630** then creates a fine-grained network policy for the updated version of the user-service component that reflects the discovered network paths from the user-service component and includes all of the component labels. An example of a fine-grained network policy created for the user-service component is shown below:

{app: user-service, zone: services, version: 1582650759}→{app: message- queue, zone: data}

In some instances, fine-grained network policy identifies the components that did not receive any internal traffic, such as api-gateway and web-app shown in the network policy **406** of FIG. **4**

The fine-grained network policy is packaged with the updated version of the user-service component into a container and deployed to a node or brought online within a node in a cluster of nodes in the containerized environment. When the new or updated version of the user-service component is deployed, its corresponding fine-grained network policy is also added to the node.

As a result, the cluster of nodes now has two versions of the user-service component, with the following labels:

Previous Version: {app: user-service, zone: services, version: 1582630603}

Updated Version: {app: user-service, zone: services, version: 1582650759}

Also, the cluster of nodes has two separate fine-grained network policies for the user-service component for the previous and updated versions of the component as shown below:

{app: user-service, zone: services, version: 1582630603}→{app: database, zone: data}

{app: user-service, zone: services, version: 1582650759}→{app: message-queue, zone: data}

Eventually, the previous version of the user-service component is retired and removed from the cluster of nodes. As a result, the corresponding network policy is also removed. At that point, the cluster of nodes contains only a single network policy for the user service component as shown below:

{app: user-service, zone: services, version: 1582650759}→{app: message- queue, zone: data}

FIG. **7** is an example of a process for generating a network policy for a component of an application executing in a containerized environment, according to certain embodiments. The processing depicted in FIG. **7** may be implemented in software only (e.g., code, instructions, program) executed by one or more processing units (e.g., processors, cores) of the respective systems, hardware only, or combinations thereof. The software may be stored on a non-transitory storage medium (e.g., on a memory device). The process **700** presented in FIG. **7** and described below is intended to be illustrative and non-limiting. Although FIG. **7** depicts the various processing steps occurring in a particular sequence or order, this is not intended to be limiting. In certain alternative embodiments, the steps may be performed in some different order or some steps may also be performed in parallel. In certain embodiments, such as in the embodiment depicted in FIG. **6**, the processing depicted in FIG. **7** may be performed by a data processing system comprising the acceptance test subsystem, the network monitoring subsystem comprising the monitoring agents, component lookup module, component monitor module, and related data stores, and the network policy creation subsystem comprising the policy creator, network policies, and related data stores.

In the embodiment depicted in FIG. **7**, processing is initiated at block **702** when the acceptance test subsystem determines/obtains a set of one or more pre-defined network security policies (i.e., coarse-grained network security polices **117**) defined for the test environment/application. As previously described, the set of pre-defined network security policies may be defined, for example, by a user of the enterprise and these policies reflect general network traffic expectations of the flow of network traffic between the components of the application in the test environment. For example, a coarse-grained policy for the order processing application **200** (shown in FIG. **2**) may be represented using "zone labels" as shown in table **402** of FIG. **4**.

At block **704**, the acceptance test subsystem receives a component to be deployed as part of a deployed application in the computing environment. As previously described, the component may be a new component of the application or an updated version of an existing component of the application for deployment in the computing environment.

At block **706**, the acceptance test subsystem performs a policy compliance test for the component. As part of performing this test, in certain examples, the acceptance test subsystem provides the component to the network monitoring subsystem which monitors the network traffic flow to and from the component at block **708**. At block **710**, the network monitoring subsystem **606** identifies a set of one or more network paths for the component based on the monitoring. In certain examples, a network path in the set of network paths identifies a source component from which the component receives packets and/or a target component to which the component sends packets. At block **712**, for each network path identified in block **712**, the acceptance test subsystem performs a policy compliance test to determine that the network path does not violate any of the coarse-grained policies determined in block **702** for the test environment/application.

In certain embodiments, in addition to the policy compliance test performed at block **706**, the acceptance test subsystem may also perform functional testing of the component **106** at block **714** to confirm that the functionality of the component is behaving as expected. The functional testing may be performed before, after, or in parallel with performing the policy compliance test for the component.

At block **716**, the acceptance test subsystem **716** performs a check to determine if the functional testing of the component passed and that the coarse-grained network security policies determined in block **702** are not violated by any network path identified in block **710**.

If the component passes the functional test and no violation of the coarse-grained policy is detected, the acceptance test subsystem deploys the component to production at block **720**. Additional details of the operations performed by the acceptance test subsystem to deploy the component to production are described below with respect to the process depicted in FIG. **8** and its accompanying description. If the component does not pass the functional test and/or a coarse-grained policy violation is detected, the acceptance test subsystem outputs information at block **718** indicating that the component should not be deployed to production.

FIG. **8** is an example of a process for detecting network connections, detecting components within a containerized environment, and generating a network policy for a component of an application executing in the containerized environment, according to another embodiment. The processing depicted in FIG. **8** describes additional details of the operations performed in block **720** of FIG. **7**. The processing depicted in FIG. **8** may be implemented in software only (e.g., code, instructions, program) executed by one or more processing units (e.g., processors, cores) of the respective systems, hardware only, or combinations thereof. The software may be stored on a non-transitory storage medium (e.g., on a memory device). The process **800** presented in FIG. **8** and described below is intended to be illustrative and non-limiting. Although FIG. **8** depicts the various processing steps occurring in a particular sequence or order, this is not intended to be limiting. In certain alternative embodiments, the steps may be performed in some different order or some steps may also be performed in parallel. In certain embodiments, such as in the embodiment depicted in FIG. **6**, the processing depicted in FIG. **8** may be performed by a data processing system comprising the network monitoring subsystem comprising the monitoring agents, component lookup module, component monitor module, and related data stores, and the network policy creation subsystem comprising the policy creator, network policies, and related data stores.

In certain embodiments, the processing depicted in the embodiment of FIG. **8** is triggered when a component is to be deployed as part of a deployed application (at block **704** of FIG. **7**) or during use of the component in a production environment (e.g., without an active policy or with only use of a course-grain policy). At block **805**, the network monitoring subsystem monitors network traffic flow originating from network interfaces corresponding to containers that execute components of an application. At block **810**, the network monitoring subsystem detects a new network connection or a change in an existing network connection within the network traffic based on the monitoring of the network traffic flow. In response to detecting the new network connection or the change in the existing network connection, at block **815**, the network monitoring subsystem records details of the new network connection or the change in the existing network connection. The details include a network address of a source component and a network address of a destination component for the new network connection or the change in the existing network connection. In some instances, the details further include a time stamp for the new network connection or the change in the existing network connection.

At block **820**, the network monitoring subsystem obtains information concerning the components of the application. The information includes the network address and metadata associated with each of the components of the application. In some instances, the information further includes any changes to arrangement of the components and time of the changes, and the metadata comprises labels associated with each of the components of the application.

At block **825**, the network monitoring subsystem identifies metadata for the source component and the destination component based on a comparison of at least the network address of the source component and the network address of the destination component to the network address associated with each of the components of the application. In some instances, the metadata comprises labels associated with each of the components of the application. The labels may be component identifiers comprising a version label that uniquely identifies the component. The version label may include, but is not limited to, timestamp, a commit identifier, or a version identifier for the component.

Optionally, the network monitoring subsystem identifies a subset of components of the components of the application that are not involved in the new network connection or the change in the existing network connection based on the comparison of at least the network address of the source component and the network address of the destination component to the network address associated with each of the components of the application.

At block **830**, the network security policy creation subsystem generates a network policy (e.g., a fine-grained network security policy (i.e., a component-specific network security policy) for the source component or the destination component using at least the metadata for the source component and the destination component. The network policy comprises information representative of the new network connection or the change in the existing network connection. The network policy may further comprise information indicating that the network policy is applicable to a defined version of the source component or the destination component. In some instances, the network policy for the source component or the destination component is generated using the metadata for the source component and the destination component, the time stamp for the new network connection or the change in the existing network connection, the time of the changes associated with the arrangement of the source component or the destination component, the subset of components of the components of the application that are not involved in the new network connection or the change in the existing network connection, or any combination thereof.

At block **835**, the network policy creation subsystem generates a deployment package comprising the source component or the destination component and the network policy associated with the source component or the destination component. The generating deployment package comprises integrating the network policy for the source component or the destination component into the deployment package for the application.

At block **840**, the network policy creation subsystem deploys the deployment package in the production environment. In certain examples, deploying the deployment package includes deploying the source component or the destination component and its associated network policy to a computing node in a cluster of nodes in a containerized environment. As previously described, in certain examples, the containerized environment may be provided by a container orchestration platform such as Kubernetes, OpenShift, Docker Swarm and the like.

In certain embodiments, the disclosed systems include capabilities for enabling different versions of a component of a containerized application to co-exist on different computing nodes in a cluster of nodes of the containerized environment at the same time. The systems additionally include capabilities for enabling different network policies to be applied to the different versions of the component, where each component has potentially different network requirements. By way of example, a first computing node in the cluster of nodes may deploy a first version of a component. The first version of the component may, for instance, be an updated version of a previous version (e.g., a second version) of the component. In certain examples, the second version of the component may be deployed to second computing node in the cluster of nodes. The first version of the component may be associated with a first network security policy where the first network security policy comprises information indicating that the first network security policy is applicable to first version of the component identified by the component identifier. As described above, the first network security policy may additionally identify at least one other component to which the first version of the component sends and receives network traffic. In certain examples, the first network security policy may also identify a zone label and an application label.

The second version of the component may be associated with a second network security policy where the second network security policy comprises information indicating that the second network security policy is applicable to the second version of the component identified by a component identifier that uniquely identifies the second version of the component. In certain embodiments, at least one network path in the set of network paths for the first version of the component may be different from a network path in a set of network paths for the second version of the component. A first deployment package comprising the first version of the component and the first network security policy is deployed to the first computing node and a second deployment package comprising the second version of the component and the second network security policy is deployed to the second computing node

FIG. **9** is an example of a process of the manner in which the network security policy of a component may be utilized, according to certain embodiments. The processing depicted in FIG. **9** may be implemented in software only (e.g., code, instructions, program) executed by one or more processing units (e.g., processors, cores) of the respective systems, hardware only, or combinations thereof. The software may be stored on a non-transitory storage medium (e.g., on a memory device). The process **900** presented in FIG. **9** and described below is intended to be illustrative and non-limiting. Although FIG. **9** depicts the various processing steps occurring in a particular sequence or order, this is not intended to be limiting. In certain alternative embodiments, the steps may be performed in some different order or some steps may also be performed in parallel. In certain embodiments, such as in the embodiment depicted in FIG. **6**, the processing depicted in FIG. **9** may be performed by the deployment orchestrator system **116** in the production environment.

At block **902**, the process involves identifying a flow of network traffic from a specific component (that is deployed as part of a deployed application) to another component of the application or the flow of network traffic from another component to the specific component. At block **904**, the process involves determining a component identifier for the component (e.g., within metadata associated with the component). At block **906**, the process involves determining a network security policy for the component based on the component identifier. At block **908**, the process involves either allowing or disallowing the flow of network traffic for the component based on the identified network security policy.

The present disclosure offers several advantages including the ability to generate network security policies for different versions of a component of an application deployed in a computing environment where the different versions have potentially different network requirements and the different versions operate together at the same time in the computing environment. The disclosed systems include capabilities for enabling different versions of a component of a containerized application to co-exist at the same time on different computing nodes in a cluster of nodes in a containerized environment that deploys and executes the application. The disclosed systems additionally include capabilities for enabling different network policies to be generated for the different versions of the component, where each component has potentially different network requirements. The disclosed systems provide a mechanism to create precise, per-component network policies, while respecting the overall coarse-grained policies of the containerized application.

Example Architectures

As noted above, infrastructure as a service (IaaS) is one particular type of cloud computing. IaaS can be configured to provide virtualized computing resources over a public network (e.g., the Internet). In an IaaS model, a cloud computing provider can host the infrastructure components (e.g., servers, storage devices, network nodes (e.g., hardware), deployment software, platform virtualization (e.g., a hypervisor layer), or the like). In some cases, an IaaS provider may also supply a variety of services to accompany those infrastructure components (e.g., billing, monitoring, logging, security, load balancing and clustering, etc.). Thus, as these services may be policy-driven, IaaS users may be able to implement policies to drive load balancing to maintain application availability and performance.

In some instances, IaaS customers may access resources and services through a wide area network (WAN), such as the Internet, and can use the cloud provider's services to install the remaining elements of an application stack. For example, the user can log in to the IaaS platform to create virtual machines (VMs), install operating systems (OSs) on each VM, deploy middleware such as databases, create storage buckets for workloads and backups, and even install enterprise software into that VM. Customers can then use the provider's services to perform various functions, including balancing network traffic, troubleshooting application issues, monitoring performance, managing disaster recovery, etc.

In most cases, a cloud computing model will require the participation of a cloud provider. The cloud provider may, but need not be, a third-party service that specializes in providing (e.g., offering, renting, selling) IaaS. An entity might also opt to deploy a private cloud, becoming its own provider of infrastructure services.

In some examples, IaaS deployment is the process of putting a new application, or a new version of an application, onto a prepared application server or the like. It may also include the process of preparing the server (e.g., installing libraries, daemons, etc.). This is often managed by the cloud provider, below the hypervisor layer (e.g., the servers, storage, network hardware, and virtualization). Thus, the customer may be responsible for handling (OS), middleware, and/or application deployment (e.g., on self-service virtual machines (e.g., that can be spun up on demand) or the like.

In some examples, IaaS provisioning may refer to acquiring computers or virtual hosts for use, and even installing needed libraries or services on them. In most cases, deployment does not include provisioning, and the provisioning may need to be performed first.

In some cases, there are two different problems for IaaS provisioning. First, there is the initial challenge of provisioning the initial set of infrastructure before anything is running. Second, there is the challenge of evolving the existing infrastructure (e.g., adding new services, changing services, removing services, etc.) once everything has been provisioned. In some cases, these two challenges may be addressed by enabling the configuration of the infrastructure to be defined declaratively. In other words, the infrastructure (e.g., what components are needed and how they interact) can be defined by one or more configuration files. Thus, the overall topology of the infrastructure (e.g., what resources depend on which, and how they each work together) can be described declaratively. In some instances, once the topology is defined, a workflow can be generated that creates and/or manages the different components described in the configuration files.

In some examples, an infrastructure may have many interconnected elements. For example, there may be one or more virtual private clouds (VPCs) (e.g., a potentially on-demand pool of configurable and/or shared computing resources), also known as a core network. In some examples, there may also be one or more security group rules provisioned to define how the security of the network will be set up and one or more virtual machines (VMs). Other infrastructure elements may also be provisioned, such as a load balancer, a database, or the like. As more and more infrastructure elements are desired and/or added, the infrastructure may incrementally evolve.

In some instances, continuous deployment techniques may be employed to enable deployment of infrastructure code across various virtual computing environments. Additionally, the described techniques can enable infrastructure management within these environments. In some examples, service teams can write code that is desired to be deployed to one or more, but often many, different production environments (e.g., across various different geographic locations, sometimes spanning the entire world). However, in some examples, the infrastructure on which the code will be deployed must first be set up. In some instances, the provisioning can be done manually, a provisioning tool may be utilized to provision the resources, and/or deployment tools may be utilized to deploy the code once the infrastructure is provisioned.

FIG. **10** is a block diagram **1000** illustrating an example pattern of an IaaS architecture, according to at least one embodiment. Service operators **1002** can be communicatively coupled to a secure host tenancy **1004** that can include a virtual cloud network (VCN) **1006** and a secure host subnet **1008**. In some examples, the service operators **1002** may be using one or more client computing devices, which may be portable handheld devices (e.g., an iPhone®, cellular telephone, an iPad®, computing tablet, a personal digital assistant (PDA)) or wearable devices (e.g., a Google Glass® head mounted display), running software such as Microsoft Windows Mobile®, and/or a variety of mobile operating systems such as iOS, Windows Phone, Android, BlackBerry 8, Palm OS, and the like, and being Internet, e-mail, short message service (SMS), Blackberry®, or other communication protocol enabled. Alternatively, the client computing devices can be general purpose personal computers including, by way of example, personal computers and/or laptop computers running various versions of Microsoft Windows®, Apple Macintosh®, and/or Linux operating systems. The client computing devices can be workstation computers running any of a variety of commercially-available UNIX® or UNIX-like operating systems, including without limitation the variety of GNU/Linux operating systems, such as for example, Google Chrome OS. Alternatively, or in addition, client computing devices may be any other electronic device, such as a thin-client computer, an Internet-enabled gaming system (e.g., a Microsoft Xbox gaming console with or without a Kinect® gesture input device), and/or a personal messaging device, capable of communicating over a network that can access the VCN **1006** and/or the Internet.

The VCN **1006** can include a local peering gateway (LPG) **1010** that can be communicatively coupled to a secure shell (SSH) VCN **1012** via an LPG **1010** contained in the SSH VCN **1012**. The SSH VCN **1012** can include an SSH subnet **1014**, and the SSH VCN **1012** can be communicatively coupled to a control plane VCN **1016** via the LPG **1010** contained in the control plane VCN **1016**. Also, the SSH VCN **1012** can be communicatively coupled to a data plane VCN **1018** via an LPG **1010**. The control plane VCN **1016** and the data plane VCN **1018** can be contained in a service tenancy **1019** that can be owned and/or operated by the IaaS provider.

The control plane VCN **1016** can include a control plane demilitarized zone (DMZ) tier **1020** that acts as a perimeter network (e.g., portions of a corporate network between the corporate intranet and external networks). The DMZ-based servers may have restricted responsibilities and help keep security breaches contained. Additionally, the DMZ tier **1020** can include one or more load balancer (LB) subnet(s) **1022**, a control plane app tier **1024** that can include app subnet(s) **1026**, a control plane data tier **1028** that can include database (DB) subnet(s) **1030** (e.g., frontend DB subnet(s) and/or backend DB subnet(s)). The LB subnet(s) **1022** contained in the control plane DMZ tier **1020** can be communicatively coupled to the app subnet(s) **1026** contained in the control plane app tier **1024** and an Internet gateway **1034** that can be contained in the control plane VCN **1016**, and the app subnet(s) **1026** can be communicatively coupled to the DB subnet(s) **1030** contained in the control plane data tier **1028** and a service gateway **1036** and a network address translation (NAT) gateway **1038**. The control plane VCN **1016** can include the service gateway **1036** and the NAT gateway **1038**.

The control plane VCN **1016** can include a data plane mirror app tier **1040** that can include app subnet(s) **1026**. The app subnet(s) **1026** contained in the data plane mirror app tier **1040** can include a virtual network interface controller (VNIC) **1042** that can execute a compute instance **1044**. The compute instance **1044** can communicatively couple the app subnet(s) **1026** of the data plane mirror app tier **1040** to app subnet(s) **1026** that can be contained in a data plane app tier **1046**.

The data plane VCN **1018** can include the data plane app tier **1046**, a data plane DMZ tier **1048**, and a data plane data tier **1050**. The data plane DMZ tier **1048** can include LB subnet(s) **1022** that can be communicatively coupled to the app subnet(s) **1026** of the data plane app tier **1046** and the Internet gateway **1034** of the data plane VCN **1018**. The app subnet(s) **1026** can be communicatively coupled to the service gateway **1036** of the data plane VCN **1018** and the NAT gateway **1038** of the data plane VCN **1018**. The data plane data tier **1050** can also include the DB subnet(s) **1030** that can be communicatively coupled to the app subnet(s) **1026** of the data plane app tier **1046**.

The Internet gateway **1034** of the control plane VCN **1016** and of the data plane VCN **1018** can be communicatively coupled to a metadata management service **1052** that can be communicatively coupled to public Internet **1054**. Public Internet **1054** can be communicatively coupled to the NAT gateway **1038** of the control plane VCN **1016** and of the data plane VCN **1018**. The service gateway **1036** of the control plane VCN **1016** and of the data plane VCN **1018** can be communicatively couple to cloud services **1056**.

In some examples, the service gateway **1036** of the control plane VCN **1016** or of the data plane VCN **1018** can make application programming interface (API) calls to cloud services **1056** without going through public Internet **1054**. The API calls to cloud services **1056** from the service gateway **1036** can be one-way: the service gateway **1036** can make API calls to cloud services **1056**, and cloud services **1056** can send requested data to the service gateway **1036**. But, cloud services **1056** may not initiate API calls to the service gateway **1036**.

In some examples, the secure host tenancy **1004** can be directly connected to the service tenancy **1019**, which may be otherwise isolated. The secure host subnet **1008** can communicate with the SSH subnet **1014** through an LPG **1010** that may enable two-way communication over an otherwise isolated system. Connecting the secure host subnet **1008** to the SSH subnet **1014** may give the secure host subnet **1008** access to other entities within the service tenancy **1019**.

The control plane VCN **1016** may allow users of the service tenancy **1019** to set up or otherwise provision desired resources. Desired resources provisioned in the control plane VCN **1016** may be deployed or otherwise used in the data plane VCN **1018**. In some examples, the control plane VCN **1016** can be isolated from the data plane VCN **1018**, and the data plane mirror app tier **1040** of the control plane VCN **1016** can communicate with the data plane app tier **1046** of the data plane VCN **1018** via VNICs **1042** that can be contained in the data plane mirror app tier **1040** and the data plane app tier **1046**.

In some examples, users of the system, or customers, can make requests, for example create, read, update, or delete (CRUD) operations, through public Internet **1054** that can communicate the requests to the metadata management service **1052**. The metadata management service **1052** can communicate the request to the control plane VCN **1016** through the Internet gateway **1034**. The request can be received by the LB subnet(s) **1022** contained in the control plane DMZ tier **1020**. The LB subnet(s) **1022** may determine that the request is valid, and in response to this determination, the LB subnet(s) **1022** can transmit the request to app subnet(s) **1026** contained in the control plane app tier **1024**. If the request is validated and requires a call to public Internet **1054**, the call to public Internet **1054** may be transmitted to the NAT gateway **1038** that can make the call to public Internet **1054**. Memory that may be desired to be stored by the request can be stored in the DB subnet(s) **1030**.

In some examples, the data plane mirror app tier **1040** can facilitate direct communication between the control plane VCN **1016** and the data plane VCN **1018**. For example, changes, updates, or other suitable modifications to configuration may be desired to be applied to the resources contained in the data plane VCN **1018**. Via a VNIC **1042**, the control plane VCN **1016** can directly communicate with, and can thereby execute the changes, updates, or other suitable modifications to configuration to, resources contained in the data plane VCN **1018**.

In some embodiments, the control plane VCN **1016** and the data plane VCN **1018** can be contained in the service tenancy **1019**. In this case, the user, or the customer, of the system may not own or operate either the control plane VCN **1016** or the data plane VCN **1018**. Instead, the IaaS provider may own or operate the control plane VCN **1016** and the data plane VCN **1018**, both of which may be contained in the service tenancy **1019**. This embodiment can enable isolation of networks that may prevent users or customers from interacting with other users', or other customers', resources. Also, this embodiment may allow users or customers of the system to store databases privately without needing to rely on public Internet **1054**, which may not have a desired level of security, for storage.

In other embodiments, the LB subnet(s) **1022** contained in the control plane VCN **1016** can be configured to receive a signal from the service gateway **1036**. In this embodiment, the control plane VCN **1016** and the data plane VCN **1018** may be configured to be called by a customer of the IaaS provider without calling public Internet **1054**. Customers of the IaaS provider may desire this embodiment since database(s) that the customers use may be controlled by the IaaS provider and may be stored on the service tenancy **1019**, which may be isolated from public Internet **1054**.

FIG. **11** is a block diagram **1100** illustrating another example pattern of an IaaS architecture, according to at least one embodiment. Service operators **1102** (e.g. service operators **1002** of FIG. **10**) can be communicatively coupled to a secure host tenancy **1104** (e.g. the secure host tenancy **1004** of FIG. **10**) that can include a virtual cloud network (VCN) **1106** (e.g. the VCN **1006** of FIG. **10**) and a secure host subnet **1108** (e.g. the secure host subnet **1008** of FIG. **10**). The VCN **1106** can include a local peering gateway (LPG) **1110** (e.g. the LPG **1010** of FIG. **10**) that can be communicatively coupled to a secure shell (SSH) VCN **1112** (e.g. the SSH VCN **1012** of FIG. **10**) via an LPG **1010** contained in the SSH VCN **1112**. The SSH VCN **1112** can include an SSH subnet **1114** (e.g. the SSH subnet **1014** of FIG. **10**), and the SSH VCN **1112** can be communicatively coupled to a control plane VCN **1116** (e.g. the control plane VCN **1016** of FIG. **10**) via an LPG **1110** contained in the control plane VCN **1116**. The control plane VCN **1116** can be contained in a service tenancy **1119** (e.g. the service tenancy **1019** of FIG. **10**), and the data plane VCN **1118** (e.g. the data plane VCN **1018** of FIG. **10**) can be contained in a customer tenancy **1121** that may be owned or operated by users, or customers, of the system.

The control plane VCN **1116** can include a control plane DMZ tier **1120** (e.g. the control plane DMZ tier **1020** of FIG. **10**) that can include LB subnet(s) **1122** (e.g. LB subnet(s) **1022** of FIG. **10**), a control plane app tier **1124** (e.g. the control plane app tier **1024** of FIG. **10**) that can include app subnet(s) **1126** (e.g. app subnet(s) **1026** of FIG. **10**), a control plane data tier **1128** (e.g. the control plane data tier **1028** of FIG. **10**) that can include database (DB) subnet(s) **1130** (e.g. similar to DB subnet(s) **1030** of FIG. **10**). The LB subnet(s) **1122** contained in the control plane DMZ tier **1120** can be communicatively coupled to the app subnet(s) **1126** contained in the control plane app tier **1124** and an Internet gateway **1134** (e.g. the Internet gateway **1034** of FIG. **10**) that can be contained in the control plane VCN **1116**, and the app subnet(s) **1126** can be communicatively coupled to the DB subnet(s) **1130** contained in the control plane data tier **1128** and a service gateway **1136** (e.g. the service gateway of FIG. **10**) and a network address translation (NAT) gateway **1138** (e.g. the NAT gateway **1038** of FIG. **10**). The control plane VCN **1116** can include the service gateway **1136** and the NAT gateway **1138**.

The control plane VCN **1116** can include a data plane mirror app tier **1140** (e.g. the data plane mirror app tier **1040** of FIG. **10**) that can include app subnet(s) **1126**. The app subnet(s) **1126** contained in the data plane mirror app tier **1140** can include a virtual network interface controller (VNIC) **1142** (e.g. the VNIC of **1042**) that can execute a compute instance **1144** (e.g. similar to the compute instance **1044** of FIG. **10**). The compute instance **1144** can facilitate communication between the app subnet(s) **1126** of the data plane mirror app tier **1140** and the app subnet(s) **1126** that can be contained in a data plane app tier **1146** (e.g. the data plane app tier **1046** of FIG. **10**) via the VNIC **1142** contained in the data plane mirror app tier **1140** and the VNIC **1142** contained in the data plane app tier **1146**.

The Internet gateway **1134** contained in the control plane VCN **1116** can be communicatively coupled to a metadata management service **1152** (e.g. the metadata management service **1052** of FIG. **10**) that can be communicatively coupled to public Internet **1154** (e.g. public Internet **1054** of FIG. **10**). Public Internet **1154** can be communicatively coupled to the NAT gateway **1138** contained in the control plane VCN **1116**. The service gateway **1136** contained in the control plane VCN **1116** can be communicatively couple to cloud services **1156** (e.g. cloud services **1056** of FIG. **10**).

In some examples, the data plane VCN **1118** can be contained in the customer tenancy **1121**. In this case, the IaaS provider may provide the control plane VCN **1116** for each customer, and the IaaS provider may, for each customer, set up a unique compute instance **1144** that is contained in the service tenancy **1119**. Each compute instance **1144** may allow communication between the control plane VCN **1116**, contained in the service tenancy **1119**, and the data plane VCN **1118** that is contained in the customer tenancy **1121**. The compute instance **1144** may allow resources, that are provisioned in the control plane VCN **1116** that is contained in the service tenancy **1119**, to be deployed or otherwise used in the data plane VCN **1118** that is contained in the customer tenancy **1121**.

In other examples, the customer of the IaaS provider may have databases that live in the customer tenancy **1121**. In this example, the control plane VCN **1116** can include the data plane mirror app tier **1140** that can include app subnet(s) **1126**. The data plane mirror app tier **1140** can reside in the data plane VCN **1118**, but the data plane mirror app tier **1140** may not live in the data plane VCN **1118**. That is, the data plane mirror app tier **1140** may have access to the customer tenancy **1121**, but the data plane mirror app tier **1140** may not exist in the data plane VCN **1118** or be owned or operated by the customer of the IaaS provider. The data plane mirror app tier **1140** may be configured to make calls to the data plane VCN **1118** but may not be configured to make calls to any entity contained in the control plane VCN **1116**. The customer may desire to deploy or otherwise use resources in the data plane VCN **1118** that are provisioned in the control plane VCN **1116**, and the data plane mirror app tier **1140** can facilitate the desired deployment, or other usage of resources, of the customer.

In some embodiments, the customer of the IaaS provider can apply filters to the data plane VCN **1118**. In this embodiment, the customer can determine what the data plane VCN **1118** can access, and the customer may restrict access to public Internet **1154** from the data plane VCN **1118**. The IaaS provider may not be able to apply filters or otherwise control access of the data plane VCN **1118** to any outside networks or databases. Applying filters and controls by the customer onto the data plane VCN **1118**, contained in the customer tenancy **1121**, can help isolate the data plane VCN **1118** from other customers and from public Internet **1154**.

In some embodiments, cloud services **1156** can be called by the service gateway **1136** to access services that may not exist on public Internet **1154**, on the control plane VCN **1116**, or on the data plane VCN **1118**. The connection between cloud services **1156** and the control plane VCN **1116** or the data plane VCN **1118** may not be live or continuous. Cloud services **1156** may exist on a different network owned or operated by the IaaS provider. Cloud services **1156** may be configured to receive calls from the service gateway **1136** and may be configured to not receive calls from public Internet **1154**. Some cloud services **1156** may be isolated from other cloud services **1156**, and the control plane VCN **1116** may be isolated from cloud services **1156** that may not be in the same region as the control plane VCN **1116**. For example, the control plane VCN **1116** may be located in "Region 1," and cloud service "Deployment 10," may be located in Region 1 and in "Region 2." If a call to Deployment 10 is made by the service gateway **1136** contained in the control plane VCN **1116** located in Region 1, the call may be transmitted to Deployment 10 in Region 1. In this example, the control plane VCN **1116**, or Deployment 10 in Region 1, may not be communicatively coupled to, or otherwise in communication with, Deployment 10 in Region 2.

FIG. **12** is a block diagram **1200** illustrating another example pattern of an IaaS architecture, according to at least one embodiment. Service operators **1202** (e.g. service operators **1002** of FIG. **10**) can be communicatively coupled to a secure host tenancy **1204** (e.g. the secure host tenancy **1004** of FIG. **10**) that can include a virtual cloud network (VCN) **1206** (e.g. the VCN **1006** of FIG. **10**) and a secure host subnet **1208** (e.g. the secure host subnet **1008** of FIG. **10**). The VCN **1206** can include an LPG **1210** (e.g. the LPG **1010** of FIG. **10**) that can be communicatively coupled to an SSH VCN **1212** (e.g. the SSH VCN **1012** of FIG. **10**) via an LPG **1210** contained in the SSH VCN **1212**. The SSH VCN **1212** can include an SSH subnet **1214** (e.g. the SSH subnet **1014** of FIG. **10**), and the SSH VCN **1212** can be communicatively coupled to a control plane VCN **1216** (e.g. the control plane VCN **1016** of FIG. **10**) via an LPG **1210** contained in the control plane VCN **1216** and to a data plane VCN **1218** (e.g. the data plane **1018** of FIG. **10**) via an LPG **1210** contained in the data plane VCN **1218**. The control plane VCN **1216** and the data plane VCN **1218** can be contained in a service tenancy **1219** (e.g. the service tenancy **1019** of FIG. **10**).

The control plane VCN **1216** can include a control plane DMZ tier **1220** (e.g. the control plane DMZ tier **1020** of FIG. **10**) that can include load balancer (LB) subnet(s) **1222** (e.g. LB subnet(s) **1022** of FIG. **10**), a control plane app tier **1224** (e.g. the control plane app tier **1024** of FIG. **10**) that can include app subnet(s) **1226** (e.g. similar to app subnet(s) **1026** of FIG. **10**), a control plane data tier **1228** (e.g. the control plane data tier **1028** of FIG. **10**) that can include DB subnet(s) **1230**. The LB subnet(s) **1222** contained in the control plane DMZ tier **1220** can be communicatively coupled to the app subnet(s) **1226** contained in the control plane app tier **1224** and to an Internet gateway **1234** (e.g. the Internet gateway **1034** of FIG. **10**) that can be contained in the control plane VCN **1216**, and the app subnet(s) **1226** can be communicatively coupled to the DB subnet(s) **1230** contained in the control plane data tier **1228** and to a service gateway **1236** (e.g. the service gateway of FIG. **10**) and a network address translation (NAT) gateway **1238** (e.g. the NAT gateway **1038** of FIG. **10**). The control plane VCN **1216** can include the service gateway **1236** and the NAT gateway **1238**.

The data plane VCN **1218** can include a data plane app tier **1246** (e.g. the data plane app tier **1046** of FIG. **10**), a data plane DMZ tier **1248** (e.g. the data plane DMZ tier **1048** of FIG. **10**), and a data plane data tier **1250** (e.g. the data plane data tier **1050** of FIG. **10**). The data plane DMZ tier **1248** can include LB subnet(s) **1222** that can be communicatively coupled to trusted app subnet(s) **1260** and untrusted app subnet(s) **1262** of the data plane app tier **1246** and the Internet gateway **1234** contained in the data plane VCN **1218**. The trusted app subnet(s) **1260** can be communicatively coupled to the service gateway **1236** contained in the data plane VCN **1218**, the NAT gateway **1238** contained in the data plane VCN **1218**, and DB subnet(s) **1230** contained in the data plane data tier **1250**. The untrusted app subnet(s) **1262** can be communicatively coupled to the service gateway **1236** contained in the data plane VCN **1218** and DB subnet(s) **1230** contained in the data plane data tier **1250**. The data plane data tier **1250** can include DB subnet(s) **1230** that can be communicatively coupled to the service gateway **1236** contained in the data plane VCN **1218**.

The untrusted app subnet(s) **1262** can include one or more primary VNICs **1264**(**1**)-(N) that can be communicatively coupled to tenant virtual machines (VMs) **1266**(**1**)-(N). Each tenant VM **1266**(**1**)-(N) can be communicatively coupled to a respective app subnet **1267**(**1**)-(N) that can be contained in respective container egress VCNs **1268**(**1**)-(N) that can be contained in respective customer tenancies **1270**(**1**)-(N). Respective secondary VNICs **1272**(**1**)-(N) can facilitate communication between the untrusted app subnet(s) **1262** contained in the data plane VCN **1218** and the app subnet contained in the container egress VCNs **1268**(**1**)-(N). Each container egress VCNs **1268**(**1**)-(N) can include a NAT gateway **1238** that can be communicatively coupled to public Internet **1254** (e.g. public Internet **1054** of FIG. **10**).

The Internet gateway **1234** contained in the control plane VCN **1216** and contained in the data plane VCN **1218** can be communicatively coupled to a metadata management service **1252** (e.g. the metadata management system **1052** of FIG. **10**) that can be communicatively coupled to public Internet **1254**. Public Internet **1254** can be communicatively coupled to the NAT gateway **1238** contained in the control plane VCN **1216** and contained in the data plane VCN **1218**. The service gateway **1236** contained in the control plane VCN **1216** and contained in the data plane VCN **1218** can be communicatively couple to cloud services **1256**.

In some embodiments, the data plane VCN **1218** can be integrated with customer tenancies **1270**. This integration can be useful or desirable for customers of the IaaS provider in some cases such as a case that may desire support when executing code. The customer may provide code to run that may be destructive, may communicate with other customer resources, or may otherwise cause undesirable effects. In response to this, the IaaS provider may determine whether to run code given to the IaaS provider by the customer.

In some examples, the customer of the IaaS provider may grant temporary network access to the IaaS provider and request a function to be attached to the data plane tier app **1246**. Code to run the function may be executed in the VMs **1266**(**1**)-(N), and the code may not be configured to run anywhere else on the data plane VCN **1218**. Each VM **1266**(**1**)-(N) may be connected to one customer tenancy **1270**. Respective containers **1271**(**1**)-(N) contained in the VMs **1266**(**1**)-(N) may be configured to run the code. In this case, there can be a dual isolation (e.g., the containers **1271**(**1**)-(N) running code, where the containers **1271**(**1**)-(N) may be contained in at least the VM **1266**(**1**)-(N) that are contained in the untrusted app subnet(s) **1262**), which may help prevent incorrect or otherwise undesirable code from damaging the network of the IaaS provider or from damaging a network of a different customer. The containers **1271** (**1**)-(N) may be communicatively coupled to the customer tenancy **1270** and may be configured to transmit or receive data from the customer tenancy **1270**. The containers **1271** (**1**)-(N) may not be configured to transmit or receive data from any other entity in the data plane VCN **1218**. Upon completion of running the code, the IaaS provider may kill or otherwise dispose of the containers **1271**(**1**)-(N).

In some embodiments, the trusted app subnet(s) **1260** may run code that may be owned or operated by the IaaS provider. In this embodiment, the trusted app subnet(s) **1260** may be communicatively coupled to the DB subnet(s) **1230** and be configured to execute CRUD operations in the DB subnet(s) **1230**. The untrusted app subnet(s) **1262** may be communicatively coupled to the DB subnet(s) **1230**, but in this embodiment, the untrusted app subnet(s) may be configured to execute read operations in the DB subnet(s) **1230**. The containers **1271**(**1**)-(N) that can be contained in the VM **1266**(**1**)-(N) of each customer and that may run code from the customer may not be communicatively coupled with the DB subnet(s) **1230**.

In other embodiments, the control plane VCN **1216** and the data plane VCN **1218** may not be directly communicatively coupled. In this embodiment, there may be no direct communication between the control plane VCN **1216** and the data plane VCN **1218**. However, communication can occur indirectly through at least one method. An LPG **1210** may be established by the IaaS provider that can facilitate communication between the control plane VCN **1216** and the data plane VCN **1218**. In another example, the control plane VCN **1216** or the data plane VCN **1218** can make a call to cloud services **1256** via the service gateway **1236**. For example, a call to cloud services **1256** from the control plane VCN **1216** can include a request for a service that can communicate with the data plane VCN **1218**.

FIG. **13** is a block diagram **1300** illustrating another example pattern of an IaaS architecture, according to at least one embodiment. Service operators **1302** (e.g. service operators **1002** of FIG. **10**) can be communicatively coupled to a secure host tenancy **1304** (e.g. the secure host tenancy **1004** of FIG. **10**) that can include a virtual cloud network (VCN) **1306** (e.g. the VCN **1006** of FIG. **10**) and a secure host subnet **1308** (e.g. the secure host subnet **1008** of FIG. **10**). The VCN **1306** can include an LPG **1310** (e.g. the LPG **1010** of FIG. **10**) that can be communicatively coupled to an SSH VCN **1312** (e.g. the SSH VCN **1012** of FIG. **10**) via an LPG **1310** contained in the SSH VCN **1312**. The SSH VCN **1312** can include an SSH subnet **1314** (e.g. the SSH subnet **1014** of FIG. **10**), and the SSH VCN **1312** can be communicatively coupled to a control plane VCN **1316** (e.g. the control plane VCN **1016** of FIG. **10**) via an LPG **1310** contained in the control plane VCN **1316** and to a data plane VCN **1318** (e.g. the data plane **1018** of FIG. **10**) via an LPG **1310** contained in the data plane VCN **1318**. The control plane VCN **1316** and the data plane VCN **1318** can be contained in a service tenancy **1319** (e.g. the service tenancy **1019** of FIG. **10**).

The control plane VCN **1316** can include a control plane DMZ tier **1320** (e.g. the control plane DMZ tier **1020** of FIG. **10**) that can include LB subnet(s) **1322** (e.g. LB subnet(s) **1022** of FIG. **10**), a control plane app tier **1324** (e.g. the control plane app tier **1024** of FIG. **10**) that can include app subnet(s) **1326** (e.g. app subnet(s) **1026** of FIG. **10**), a control plane data tier **1328** (e.g. the control plane data tier **1028** of FIG. **10**) that can include DB subnet(s) **1330** (e.g. DB subnet(s) **1230** of FIG. **12**). The LB subnet(s) **1322** contained in the control plane DMZ tier **1320** can be communicatively coupled to the app subnet(s) **1326** contained in the control plane app tier **1324** and to an Internet gateway **1334** (e.g. the Internet gateway **1034** of FIG. **10**) that can be contained in the control plane VCN **1316**, and the app subnet(s) **1326** can be communicatively coupled to the DB subnet(s) **1330** contained in the control plane data tier **1328** and to a service gateway **1336** (e.g. the service gateway of FIG. **10**) and a network address translation (NAT) gateway **1338** (e.g. the NAT gateway **1038** of FIG. **10**). The control plane VCN **1316** can include the service gateway **1336** and the NAT gateway **1338**.

The data plane VCN **1318** can include a data plane app tier **1346** (e.g. the data plane app tier **1046** of FIG. **10**), a data plane DMZ tier **1348** (e.g. the data plane DMZ tier **1048** of FIG. **10**), and a data plane data tier **1350** (e.g. the data plane data tier **1050** of FIG. **10**). The data plane DMZ tier **1348** can include LB subnet(s) **1322** that can be communicatively coupled to trusted app subnet(s) **1360** (e.g. trusted app subnet(s) **1260** of FIG. **12**) and untrusted app subnet(s) **1362** (e.g. untrusted app subnet(s) **1262** of FIG. **12**) of the data plane app tier **1346** and the Internet gateway **1334** contained in the data plane VCN **1318**. The trusted app subnet(s) **1360** can be communicatively coupled to the service gateway **1336** contained in the data plane VCN **1318**, the NAT gateway **1338** contained in the data plane VCN **1318**, and DB subnet(s) **1330** contained in the data plane data tier **1350**. The untrusted app subnet(s) **1362** can be communicatively coupled to the service gateway **1336** contained in the data plane VCN **1318** and DB subnet(s) **1330** contained in the data plane data tier **1350**. The data plane data tier **1350** can include DB subnet(s) **1330** that can be communicatively coupled to the service gateway **1336** contained in the data plane VCN **1318**.

The untrusted app subnet(s) **1362** can include primary VNICs **1364**(**1**)-(N) that can be communicatively coupled to tenant virtual machines (VMs) **1366**(**1**)-(N) residing within the untrusted app subnet(s) **1362**. Each tenant VM **1366**(**1**)-(N) can run code in a respective container **1367**(**1**)-(N), and be communicatively coupled to an app subnet **1326** that can be contained in a data plane app tier **1346** that can be contained in a container egress VCN **1368**. Respective secondary VNICs **1372**(**1**)-(N) can facilitate communication between the untrusted app subnet(s) **1362** contained in the data plane VCN **1318** and the app subnet contained in the container egress VCN **1368**. The container egress VCN can include a NAT gateway **1338** that can be communicatively coupled to public Internet **1354** (e.g. public Internet **1054** of FIG. **10**).

The Internet gateway **1334** contained in the control plane VCN **1316** and contained in the data plane VCN **1318** can be communicatively coupled to a metadata management service **1352** (e.g. the metadata management system **1052** of FIG. **10**) that can be communicatively coupled to public Internet **1354**. Public Internet **1354** can be communicatively coupled to the NAT gateway **1338** contained in the control plane VCN **1316** and contained in the data plane VCN **1318**. The service gateway **1336** contained in the control plane VCN **1316** and contained in the data plane VCN **1318** can be communicatively couple to cloud services **1356**.

In some examples, the pattern illustrated by the architecture of block diagram **1300** of FIG. **13** may be considered an exception to the pattern illustrated by the architecture of block diagram **1200** of FIG. **12** and may be desirable for a customer of the IaaS provider if the IaaS provider cannot directly communicate with the customer (e.g., a disconnected region). The respective containers **1367**(**1**)-(N) that are contained in the VMs **1366**(**1**)-(N) for each customer can be accessed in real-time by the customer. The containers **1367**(**1**)-(N) may be configured to make calls to respective secondary VNICs **1372**(**1**)-(N) contained in app subnet(s) **1326** of the data plane app tier **1346** that can be contained in the container egress VCN **1368**. The secondary VNICs **1372**(**1**)-(N) can transmit the calls to the NAT gateway **1338** that may transmit the calls to public Internet **1354**. In this example, the containers **1367**(**1**)-(N) that can be accessed in real-time by the customer can be isolated from the control plane VCN **1316** and can be isolated from other entities contained in the data plane VCN **1318**. The containers **1367**(**1**)-(N) may also be isolated from resources from other customers.

In other examples, the customer can use the containers **1367**(**1**)-(N) to call cloud services **1356**. In this example, the customer may run code in the containers **1367**(**1**)-(N) that requests a service from cloud services **1356**. The containers **1367**(**1**)-(N) can transmit this request to the secondary VNICs **1372**(**1**)-(N) that can transmit the request to the NAT gateway that can transmit the request to public Internet **1354**. Public Internet **1354** can transmit the request to LB subnet(s) **1322** contained in the control plane VCN **1316** via the Internet gateway **1334**. In response to determining the request is valid, the LB subnet(s) can transmit the request to app subnet(s) **1326** that can transmit the request to cloud services **1356** via the service gateway **1336**.

It should be appreciated that IaaS architectures **1000**, **1100**, **1200**, **1300** depicted in the figures may have other components than those depicted. Further, the embodiments shown in the figures are only some examples of a cloud infrastructure system that may incorporate an embodiment of the disclosure. In some other embodiments, the IaaS systems may have more or fewer components than shown in the figures, may combine two or more components, or may have a different configuration or arrangement of components.

In certain embodiments, the IaaS systems described herein may include a suite of applications, middleware, and database service offerings that are delivered to a customer in a self-service, subscription-based, elastically scalable, reliable, highly available, and secure manner. An example of such an IaaS system is the Oracle Cloud Infrastructure (OCI) provided by the present assignee.

FIG. **14** illustrates an example computer system **1400**, in which various embodiments may be implemented. The system **1400** may be used to implement any of the computer systems described above. As shown in the figure, computer system **1400** includes a processing unit **1404** that communicates with a number of peripheral subsystems via a bus subsystem **1402**. These peripheral subsystems may include a processing acceleration unit **1406**, an I/O subsystem **1408**, a storage subsystem **1418** and a communications subsystem **1424**. Storage subsystem **1418** includes tangible computer-readable storage media **1422** and a system memory **1410**.

Bus subsystem **1402** provides a mechanism for letting the various components and subsystems of computer system **1400** communicate with each other as intended. Although bus subsystem **1402** is shown schematically as a single bus, alternative embodiments of the bus subsystem may utilize multiple buses. Bus subsystem **1402** may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. For example, such architectures may include an Industry Standard Architecture (ISA) bus, Micro Channel Architecture (MCA) bus, Enhanced ISA (EISA) bus, Video Electronics Standards Association (VESA) local bus, and Peripheral Component Interconnect (PCI) bus, which can be implemented as a Mezzanine bus manufactured to the IEEE P1386.1 standard.

Processing unit **1404**, which can be implemented as one or more integrated circuits (e.g., a conventional microprocessor or microcontroller), controls the operation of computer system **1400**. One or more processors may be included in processing unit **1404**. These processors may include single core or multicore processors. In certain embodiments, processing unit **1404** may be implemented as one or more independent processing units **1432** and/or **1434** with single or multicore processors included in each processing unit. In other embodiments, processing unit **1404** may also be implemented as a quad-core processing unit formed by integrating two dual-core processors into a single chip.

In various embodiments, processing unit **1404** can execute a variety of programs in response to program code and can maintain multiple concurrently executing programs or processes. At any given time, some or all of the program code to be executed can be resident in processor(s) **1404** and/or in storage subsystem **1418**. Through suitable programming, processor(s) **1404** can provide various functionalities described above. Computer system **1400** may additionally include a processing acceleration unit **1406**, which can include a digital signal processor (DSP), a special-purpose processor, and/or the like.

I/O subsystem **1408** may include user interface input devices and user interface output devices. User interface input devices may include a keyboard, pointing devices such as a mouse or trackball, a touchpad or touch screen incorporated into a display, a scroll wheel, a click wheel, a dial, a button, a switch, a keypad, audio input devices with voice command recognition systems, microphones, and other types of input devices. User interface input devices may include, for example, motion sensing and/or gesture recognition devices such as the Microsoft Kinect® motion sensor that enables users to control and interact with an input device, such as the Microsoft Xbox® 360 game controller, through a natural user interface using gestures and spoken commands. User interface input devices may also include eye gesture recognition devices such as the Google Glass® blink detector that detects eye activity (e.g., 'blinking' while taking pictures and/or making a menu selection) from users and transforms the eye gestures as input into an input device (e.g., Google Glass®). Additionally, user interface input devices may include voice recognition sensing devices that enable users to interact with voice recognition systems (e.g., Siri® navigator), through voice commands.

User interface input devices may also include, without limitation, three dimensional (3D) mice, joysticks or pointing sticks, gamepads and graphic tablets, and audio/visual devices such as speakers, digital cameras, digital camcorders, portable media players, webcams, image scanners, fingerprint scanners, barcode reader 3D scanners, 3D printers, laser rangefinders, and eye gaze tracking devices. Additionally, user interface input devices may include, for example, medical imaging input devices such as computed tomography, magnetic resonance imaging, position emission tomography, medical ultrasonography devices. User interface input devices may also include, for example, audio input devices such as MIDI keyboards, digital musical instruments and the like.

User interface output devices may include a display subsystem, indicator lights, or non-visual displays such as audio output devices, etc. The display subsystem may be a cathode ray tube (CRT), a flat-panel device, such as that using a liquid crystal display (LCD) or plasma display, a projection device, a touch screen, and the like. In general, use of the term "output device" is intended to include all possible types of devices and mechanisms for outputting information from computer system **1400** to a user or other computer. For example, user interface output devices may include, without limitation, a variety of display devices that visually convey text, graphics and audio/video information such as monitors, printers, speakers, headphones, automotive navigation systems, plotters, voice output devices, and modems.

Computer system **1400** may comprise a storage subsystem **1418** that comprises software elements, shown as being currently located within a system memory **1410**. System memory **1410** may store program instructions that are loadable and executable on processing unit **1404**, as well as data generated during the execution of these programs.

Depending on the configuration and type of computer system **1400**, system memory **1410** may be volatile (such as random access memory (RAM)) and/or non-volatile (such as read-only memory (ROM), flash memory, etc.) The RAM typically contains data and/or program modules that are immediately accessible to and/or presently being operated and executed by processing unit **1404**. In some implementations, system memory **1410** may include multiple different types of memory, such as static random access memory (SRAM) or dynamic random access memory (DRAM). In some implementations, a basic input/output system (BIOS), containing the basic routines that help to transfer information between elements within computer system **1400**, such as during start-up, may typically be stored in the ROM. By way of example, and not limitation, system memory **1410** also illustrates application programs **1412**, which may include client applications, Web browsers, mid-tier applications, relational database management systems (RDBMS), etc., program data **1414**, and an operating system **1416**. By way of example, operating system **1416** may include various versions of Microsoft Windows®, Apple Macintosh®, and/or Linux operating systems, a variety of commercially-available UNIX® or UNIX-like operating systems (including without limitation the variety of GNU/Linux operating systems, the Google Chrome® OS, and the like) and/or mobile operating systems such as iOS, Windows® Phone, Android® OS, BlackBerry® 14 OS, and Palm® OS operating systems.

Storage subsystem **1418** may also provide a tangible computer-readable storage medium for storing the basic programming and data constructs that provide the functionality of some embodiments. Software (programs, code modules, instructions) that when executed by a processor provide the functionality described above may be stored in storage subsystem **1418**. These software modules or instructions may be executed by processing unit **1404**. Storage subsystem **1418** may also provide a repository for storing data used in accordance with the present disclosure.

Storage subsystem **1400** may also include a computer-readable storage media reader **1420** that can further be connected to computer-readable storage media **1422**. Together and, optionally, in combination with system memory **1410**, computer-readable storage media **1422** may comprehensively represent remote, local, fixed, and/or removable storage devices plus storage media for temporarily and/or more permanently containing, storing, transmitting, and retrieving computer-readable information.

Computer-readable storage media **1422** containing code, or portions of code, can also include any appropriate media known or used in the art, including storage media and communication media, such as but not limited to, volatile and non-volatile, removable and non-removable media implemented in any method or technology for storage and/or transmission of information. This can include tangible computer-readable storage media such as RAM, ROM, electronically erasable programmable ROM (EEPROM), flash memory or other memory technology, CD-ROM, digital versatile disk (DVD), or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or other tangible computer readable media. This can also include nontangible computer-readable media, such as data signals, data transmissions, or any other medium which can be used to transmit the desired information and which can be accessed by computing system **1400**.

By way of example, computer-readable storage media **1422** may include a hard disk drive that reads from or writes to non-removable, nonvolatile magnetic media, a magnetic disk drive that reads from or writes to a removable, nonvolatile magnetic disk, and an optical disk drive that reads from or writes to a removable, nonvolatile optical disk such as a CD ROM, DVD, and Blu-Ray® disk, or other optical media. Computer-readable storage media **1422** may include, but is not limited to, Zip® drives, flash memory cards, universal serial bus (USB) flash drives, secure digital (SD) cards, DVD disks, digital video tape, and the like. Computer-readable storage media **1422** may also include, solid-state drives (SSD) based on non-volatile memory such as flash-memory based SSDs, enterprise flash drives, solid state ROM, and the like, SSDs based on volatile memory such as solid state RAM, dynamic RAM, static RAM, DRAM-based SSDs, magnetoresistive RAM (MRAM) SSDs, and hybrid SSDs that use a combination of DRAM and flash memory based SSDs. The disk drives and their associated computer-readable media may provide non-volatile storage of computer-readable instructions, data structures, program modules, and other data for computer system **1400**.

Communications subsystem **1424** provides an interface to other computer systems and networks. Communications subsystem **1424** serves as an interface for receiving data from and transmitting data to other systems from computer system **1400**. For example, communications subsystem **1424** may enable computer system **1400** to connect to one or more devices via the Internet. In some embodiments communications subsystem **1424** can include radio frequency (RF) transceiver components for accessing wireless voice and/or data networks (e.g., using cellular telephone technology, advanced data network technology, such as 3G, 4G or EDGE (enhanced data rates for global evolution), WiFi (IEEE 802.11 family standards, or other mobile communication technologies, or any combination thereof), global positioning system (GPS) receiver components, and/or other components. In some embodiments communications subsystem **1424** can provide wired network connectivity (e.g., Ethernet) in addition to or instead of a wireless interface.

In some embodiments, communications subsystem **1424** may also receive input communication in the form of structured and/or unstructured data feeds **1426**, event streams **1428**, event updates **1430**, and the like on behalf of one or more users who may use computer system **1400**.

By way of example, communications subsystem **1424** may be configured to receive data feeds **1426** in real-time from users of social networks and/or other communication services such as Twitter® feeds, Facebook® updates, web feeds such as Rich Site Summary (RSS) feeds, and/or real-time updates from one or more third party information sources.

Additionally, communications subsystem **1424** may also be configured to receive data in the form of continuous data streams, which may include event streams **1428** of real-time events and/or event updates **1430**, that may be continuous or unbounded in nature with no explicit end. Examples of applications that generate continuous data may include, for example, sensor data applications, financial tickers, network performance measuring tools (e.g. network monitoring and traffic management applications), clickstream analysis tools, automobile traffic monitoring, and the like.

Communications subsystem **1424** may also be configured to output the structured and/or unstructured data feeds **1426**, event streams **1428**, event updates **1430**, and the like to one or more databases that may be in communication with one or more streaming data source computers coupled to computer system **1400**.

Computer system **1400** can be one of various types, including a handheld portable device (e.g., an iPhone® cellular phone, an iPad® computing tablet, a PDA), a wearable device (e.g., a Google Glass® head mounted display), a PC, a workstation, a mainframe, a kiosk, a server rack, or any other data processing system.

Due to the ever-changing nature of computers and networks, the description of computer system **1400** depicted in the figure is intended only as a specific example. Many other configurations having more or fewer components than the system depicted in the figure are possible. For example, customized hardware might also be used and/or particular elements might be implemented in hardware, firmware, software (including applets), or a combination. Further, connection to other computing devices, such as network input/output devices, may be employed. Based on the disclosure and teachings provided herein, a person of ordinary skill in the art will appreciate other ways and/or methods to implement the various embodiments.

Although specific embodiments have been described, various modifications, alterations, alternative constructions, and equivalents are also encompassed within the scope of the disclosure. Embodiments are not restricted to operation within certain specific data processing environments, but are free to operate within a plurality of data processing environments. Additionally, although embodiments have been described using a particular series of transactions and steps, it should be apparent to those skilled in the art that the scope of the present disclosure is not limited to the described series of transactions and steps. Various features and aspects of the above-described embodiments may be used individually or jointly.

Further, while embodiments have been described using a particular combination of hardware and software, it should be recognized that other combinations of hardware and software are also within the scope of the present disclosure. Embodiments may be implemented only in hardware, or only in software, or using combinations thereof. The various processes described herein can be implemented on the same processor or different processors in any combination. Accordingly, where components or modules are described as being configured to perform certain operations, such configuration can be accomplished, e.g., by designing electronic circuits to perform the operation, by programming programmable electronic circuits (such as microprocessors) to perform the operation, or any combination thereof. Processes can communicate using a variety of techniques including but not limited to conventional techniques for inter process communication, and different pairs of processes may use different techniques, or the same pair of processes may use different techniques at different times.

The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense. It will, however, be evident that additions, subtractions, deletions, and other modifications and changes may be made thereunto without departing from the broader spirit and scope as set forth in the claims. Thus, although specific disclosure embodiments have been described, these are not intended to be limiting. Various modifications and equivalents are within the scope of the following claims.

The use of the terms "a" and "an" and "the" and similar referents in the context of describing the disclosed embodiments (especially in the context of the following claims) are to be construed to cover both the singular and the plural, unless otherwise indicated herein or clearly contradicted by context. The terms "comprising," "having," "including," and "containing" are to be construed as open-ended terms (i.e., meaning "including, but not limited to,") unless otherwise noted. The term "connected" is to be construed as partly or wholly contained within, attached to, or joined together, even if there is something intervening. Recitation of ranges of values herein are merely intended to serve as a shorthand method of referring individually to each separate value falling within the range, unless otherwise indicated herein and each separate value is incorporated into the specification as if it were individually recited herein. All methods described herein can be performed in any suitable order unless otherwise indicated herein or otherwise clearly contradicted by context. The use of any and all examples, or exemplary language (e.g., "such as") provided herein, is intended merely to better illuminate embodiments and does not pose a limitation on the scope of the disclosure unless otherwise claimed. No language in the specification should be construed as indicating any non-claimed element as essential to the practice of the disclosure.

Disjunctive language such as the phrase "at least one of X, Y, or Z," unless specifically stated otherwise, is intended to be understood within the context as used in general to present that an item, term, etc., may be either X, Y, or Z, or any combination thereof (e.g., X, Y, and/or Z). Thus, such disjunctive language is not generally intended to, and should not, imply that certain embodiments require at least one of X, at least one of Y, or at least one of Z to each be present.

Preferred embodiments of this disclosure are described herein, including the best mode known for carrying out the disclosure. Variations of those preferred embodiments may become apparent to those of ordinary skill in the art upon reading the foregoing description. Those of ordinary skill should be able to employ such variations as appropriate and the disclosure may be practiced otherwise than as specifically described herein. Accordingly, this disclosure includes all modifications and equivalents of the subject matter recited in the claims appended hereto as permitted by applicable law. Moreover, any combination of the above-described elements in all possible variations thereof is encompassed by the disclosure unless otherwise indicated herein.

All references, including publications, patent applications, and patents, cited herein are hereby incorporated by reference to the same extent as if each reference were individually and specifically indicated to be incorporated by reference and were set forth in its entirety herein.

In the foregoing specification, aspects of the disclosure are described with reference to specific embodiments thereof, but those skilled in the art will recognize that the disclosure is not limited thereto. Various features and aspects of the above-described disclosure may be used individually or jointly. Further, embodiments can be utilized in any number of environments and applications beyond those described herein without departing from the broader spirit and scope of the specification. The specification and drawings are, accordingly, to be regarded as illustrative rather than restrictive.

What is claimed is:

1. A method comprising:

monitoring, by a data processing system, network traffic flow originating from network interfaces corresponding to containers that execute components of an application;

detecting, by the data processing system, a new network connection or a change in an existing network connection within the network traffic flow based on the monitoring of the network traffic flow;

in response to detecting the new network connection or the change in the existing network connection, recording, by the data processing system, details of the new network connection or the change in the existing network connection, wherein the details include a network address of a source component and a network address of a destination component for the new network connection or the change in the existing network connection;

obtaining, by the data processing system, information concerning the components of the application, wherein the information includes the network address and metadata associated with each of the components of the application;

identifying, by the data processing system, metadata for the source component and the destination component based on a comparison of at least the network address of the source component and the network address of the destination component to the network address associated with each of the components of the application;

generating, by the data processing system, a network policy for the source component or the destination component using at least the metadata for the source component and the destination component, wherein the network policy comprises information representative of the new network connection or the change in the existing network connection; and

integrating, by the data processing system, the network policy for the source component or the destination component into a deployment package for the application.

2. The method of claim **1**, wherein the details further include a time stamp for the new network connection or the change in the existing network connection.

3. The method of claim **2**, wherein the information further includes any changes to arrangement of the components and time of the changes, and the metadata comprises labels associated with each of the components of the application.

4. The method of claim **3**, wherein the network policy for the source component or the destination component is generated using at least the metadata for the source component and the destination component, the time stamp for the new network connection or the change in the existing network connection, and the time of the changes associated with the arrangement of the source component or the destination component.

5. The method of claim **1**, further comprising identifying, by the data processing system, a subset of components of the components of the application that are not involved in the new network connection or the change in the existing network connection based on the comparison of at least the network address of the source component and the network address of the destination component to the network address associated with each of the components of the application, wherein the network policy for the source component or the destination component is generated using at least the metadata for the source component and the destination component and the subset of components of the components of the application that are not involved in the new network connection or the change in the existing network connection.

6. The method of claim **1**, wherein the network policy comprises information indicating that the network policy is applicable to a defined version of the source component or the destination component.

7. The method of claim **1**, further comprising deploying, by the data processing system, the deployment package to a computing node in a computing environment of the data processing system.

8. A system comprising:

a processor; and

a memory storing instructions that, when executed by the processor, configure the system to:

monitor network traffic flow originating from network interfaces corresponding to containers that execute components of an application;

detect a new network connection or a change in an existing network connection within the network traffic flow based on the monitoring of the network traffic flow;

in response to detecting the new network connection or the change in the existing network connection, record details of the new network connection or the change in the existing network connection, wherein the details include a network address of a source component and a network address of a destination component for the new network connection or the change in the existing network connection;

obtain information concerning the components of the application, wherein the information includes the network address and metadata associated with each of the components of the application;

identify metadata for the source component and the destination component based on a comparison of at least the network address of the source component and the network address of the destination component to the network address associated with each of the components of the application;

generate a network policy for the source component or the destination component using at least the metadata for the source component and the destination component, wherein the network policy comprises information representative of the new network connection or the change in the existing network connection; and

integrate the network policy for the source component or the destination component into a deployment package for the application.

9. The system of claim **8**, wherein the details further include a time stamp for the new network connection or the change in the existing network connection.

10. The system of claim **9**, wherein the information further includes any changes to arrangement of the components and time of the changes, and the metadata comprises labels associated with each of the components of the application.

11. The system of claim **10**, wherein the network policy for the source component or the destination component is generated using at least the metadata for the source component and the destination component, the time stamp for the new network connection or the change in the existing network connection, and the time of the changes associated with the arrangement of the source component or the destination component.

12. The system of claim **8**, wherein the system is further configured to identify a subset of components of the components of the application that are not involved in the new network connection or the change in the existing network connection based on the comparison of at least the network address of the source component and the network address of the destination component to the network address associated with each of the components of the application, wherein the network policy for the source component or the destination component is generated using at least the metadata for the source component and the destination component and the subset of components of the components of the application that are not involved in the new network connection or the change in the existing network connection.

13. The system of claim **8**, wherein the network policy comprises information indicating that the network policy is applicable to a defined version of the source component or the destination component.

14. The system of claim **8**, wherein the system is further configured to deploy the deployment package to a computing node in a computing environment of the system.

15. A non-transitory computer-readable medium having program code that is stored thereon, the program code executable by one or more processing devices for performing operations comprising:

monitoring network traffic flow originating from network interfaces corresponding to containers that execute components of an application;

detecting a new network connection or a change in an existing network connection within the network traffic flow based on the monitoring of the network traffic flow;

in response to detecting the new network connection or the change in the existing network connection, recording details of the new network connection or the change in the existing network connection, wherein the details include a network address of a source component and a network address of a destination component for the new network connection or the change in the existing network connection;

obtaining information concerning the components of the application, wherein the information includes the network address and metadata associated with each of the components of the application;

identifying metadata for the source component and the destination component based on a comparison of at least the network address of the source component and the network address of the destination component to the network address associated with each of the components of the application;

generating a network policy for the source component or the destination component using at least the metadata for the source component and the destination component, wherein the network policy comprises information representative of the new network connection or the change in the existing network connection; and

integrating the network policy for the source component or the destination component into a deployment package for the application.

16. The non-transitory computer-readable medium of claim **15**, wherein the details further include a time stamp for the new network connection or the change in the existing network connection.

17. The non-transitory computer-readable medium of claim **16**, wherein the information further includes any changes to arrangement of the components and time of the changes, and the metadata comprises labels associated with each of the components of the application.

18. The non-transitory computer-readable medium of claim **17**, wherein the network policy for the source component or the destination component is generated using at least the metadata for the source component and the destination component, the time stamp for the new network connection or the change in the existing network connection, and the time of the changes associated with the arrangement of the source component or the destination component.

19. The non-transitory computer-readable medium of claim **15**, wherein the operations further comprise identifying a subset of components of the components of the application that are not involved in the new network connection or the change in the existing network connection based on the comparison of at least the network address of the source component and the network address of the destination component to the network address associated with each of the components of the application, wherein the network policy for the source component or the destination component is generated using at least the metadata for the source component and the destination component and the subset of components of the components of the application that are not involved in the new network connection or the change in the existing network connection.

20. The non-transitory computer-readable medium of claim **15**, wherein the network policy comprises information indicating that the network policy is applicable to a defined version of the source component or the destination component.

* * * * *